UNIVERSITY OF THESSALY

DIPLOMA THESIS

Visual Perception, State Estimation & Automated Control for Self-Driving Cars

---

Author:
Eleftherios Panagiotis
LOUKAS

Supervisors:
Panagiota
TSOMPANOPOULOU
Nikolaos BELLAS
Georgios STAMOULIS

A thesis submitted in fulfillment of the requirements
for the degree of Diploma Thesis

in the

Volos, Greece, June 2019

"If the game shakes me or breaks me, I hope it makes me a better man."

Christopher G. Wallace

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Οπτική Αντίληψη, Εκτίμηση Κατάστασης & Αυτόματος Έλεγχος για Αυτοκινούμενα Οχήματα

Ελευθέριος Παναγιώτης Λούκας

Η πολυεκατομμυριούχος βιομηχανία των αυτοκινούμενων οχημάτων πρόκειται να φτάσει στις ζωές μας και η πρόκληση της αυτονομίας απαιτεί πολύχρονη έρευνα από κορυφαία ιδρύματα και επιχειρήσεις. Η εργασία αυτή διερευνά και αναλύει τις σύγχρονες τεχνικές που χρησιμοποιούνται στα εφαρμοσμένα συστήματα ελέγχου, την εκτίμηση της κατάστασης και την οπτική αντίληψη για τα οχήματα χωρίς οδηγό. Το περιβάλλον προσομοίωσης CARLA χρησιμοποιείται για γνήσια δημιουργία δεδομένων. Αρχικά, παρουσιάζουμε τον τρόπο με τον οποίο οι αυτόματοι ελεγκτές PID και Stanley μπορούν να αυτοματοποιήσουν την κίνηση και το σύστημα πορείας. Έπειτα, συγχωνεύουμε διάφορους αισθητήρες, όπως το IMU, το LIDAR και το GPS, εφαρμόζοντας το Extended Kalman Filter, προκειμένου να εκτιμήσουμε αποτελεσματικά την κατάσταση και την τοποθεσία ενός έξυπνου αυτοκινήτου. Στη συνέχεια, επεκτείνουμε την έρευνα ανιχνεύοντας την κίνηση της κάμερας μεταξύ διαδοχικών εικόνων, επιτυγχάνοντας οπτική οδομετρία. Τέλος, χρησιμοποιούμε το βαθύ νευρωνικό δίκτυο κατάτμησης του CARLA και το συνδυάζουμε με κλασσικές τεχνικές όρασης υπολογιστών προκειμένου να κατανοήσουμε με ακρίβεια την οδική σκηνή για διάφορα σενάρια όπως την εξίσωση επιπέδου του δρόμου, την εκτίμηση λωρίδων και υλοποιούμε ένα σύστημα σύγκρουσης. Τα ευρήματα αυτής της μελέτης παρέχουν μια δομημένη προσέγγιση για την επίλυση τέτοιων προβλημάτων και μπορεί να υποδεικνύουν μεγάλη σημασία για μηχανικούς έξυπνων συστημάτων.

Λέξεις-Κλειδιά: αυτοκινούμενα οχήματα, όραση υπολογιστών, *CARLA*, τεχνητή νοημοσύνη, βαθιά μάθηση, μηχανική μάθηση, αυτόνομα αμάξια, συστήματα αυτομάτου ελέγχου·

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Diploma Thesis

Visual Perception, State Estimation & Automated Control for Self-Driving Cars

by Eleftherios Panagiotis LOUKAS

The multi-billion dollar industry of self-driving cars has arrived, and the challenge of autonomy requires a vast amount of research from top institutions and enterprises. This thesis explores and analyzes the state-of-the-art engineering practices used in applied control systems, state estimation, and visual perception for driverless vehicles. The CARLA Simulation Environment is exploited for genuine data generation. At first, we investigate how PID and Stanley controllers can automate the driving task of throttling, braking, and steering. Then, we fuse various sensors such as IMU, LIDAR, and GPS by applying the Extended Kalman Filter in order to estimate a smart car's state and location efficiently. Next, the investigation is extended by tracking the camera motion between frames, achieving visual odometry. Finally, we leverage CARLA's deep segmentation neural network and combine it with classical computer vision techniques in order to accomplish accurate road scene understanding for different scenarios: ground plane fitting, lane estimation and an example collision system. The findings of this study provide a structured approach to the solution of such problems and may indicate importance to intelligent systems engineers.

Keywords: *self-driving cars, computer vision, CARLA, artificial intelligence, deep learning, machine learning, autonomous vehicles, automated control systems;*

Acknowledgements

First and foremost, I would like to show appreciation to my supervisor Prof. Yota Tsompanopoulou for giving me the chance to work on this thesis, which is a project that I wanted to work since a long time. Her availability, friendliness and guidance had a prominent role to the development of this thesis and not only. She trusted me and helped me grow in many scenarios during my university years and I would like to render my warmest thanks to her. I would also wish to express my gratitude to my advisors Prof. Nikolaos Bellas, for his assistance and all the discussions we had in his office, and Prof. Georgios Stamoulis, for his cordiality and kindness in all of these years.

In addition, I would like to thank Prof. Emeritus Elias N. Houstis for his invaluable support during these years and the motivation that he helped me have in order to study the inspiring field of data science and machine learning.

Furthermore, I am always grateful to my family for their continuous and unconditional support. They helped me become a strong independent person, always with the will to discuss, travel and explore.

Last but not least, I would like to wish the best to all my 'old' friends from my hometown Larissa, the 'international' ones that I made in Lisbon and especially all my 'new' buddies in Volos. Each one of them was there to help me in my worst and best moods during these five years and they hold a significant place in my heart. I truly hope that we will stay connected no matter what happens and that everyone will chase and win their dreams.

Especially, I thank Valina, for all her love and support all this time.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The transportation system nowadays could be considered broken. More than one million people die every year due to manual driving car accidents. Sometimes, people are driving drunk or drugged. However, even while sober, the human's eye distraction is enough for a fatal car accident to occur. Traffic is also another big contributing factor to this statement, a problem occurring almost in any big city. The driving ecosystem is getting revolutionized by two types of companies: shared-economy enterprises like Lyft or Uber, and autonomous car manufacturers like Tesla or Waymo. Such companies have felt the need for eliminating car ownership and increasing access to mobility while building the automated future of transportation, which needs to be personalized, reliable, and money-saving. In conclusion, vehicles should be fully autonomous, eliminating any risk of danger, in contrast to manual driving, having a substantial impact on safety, equity, and environmental issues in our society.

Many enterprises are working on large-scale deployments, such as Waymo's robotaxis, Amazon's driverless delivery, and Tesla's Autopilot. The industry of driverless cars has accomplished many milestones throughout these years, and more of it is still expected. For sure, the era of machine and deep learning is one major contributing factor to this development. The amount of data produced is getting multiplied every single day. In fact, 90% of the world's data has been generated over the last two years. Of course, this is due to the massive and mainstream applications of the internet in our everyday lives. Services that are data-correlated include social media, smart homes, cloud storages, and more. Internet-Of-Things (IoT) is rapidly accelerating the growth of the data and information that is easily accessible, which can be fed into machine learning algorithms in order to produce outputs. So, the era of deep learning and its relation to the typical IoT applications have revolutionized the Artificial Intelligence field, and surely, it is one big player in the self-driving car industry, where market researchers predict more than twenty million driverless vehicles on the road by 2025.

However, such a future should not be considered 'utopian'. AI systems might sometimes fail under non-human interpretable controls. Also, they might be biased in non-ethical social ways. For example, security and personal information should be protected in every way. Engineers need to consider many aspects, both technical and social or political ones, before developing a system like this.

The self-driving car industry promises a vast income generation for the societies, consisting of many billions of dollars while reducing the number of fatal accidents taking place. Besides, it is transforming the way people travel, making them no more responsible for steering, throttling, braking or detecting possible danger, allowing them to spend their free time in any way they desire during a car journey. This can be achieved through selective processing of a smart car's information, which gets collected through its sensors, one of its key components.

1.2 Thesis Statement and Contributions

Researchers from leading organizations contribute solutions and findings for many problems and questions that are related to the driverless vehicle 'idea', every single day.

There are many questions that this research-driven community tries to answer, like:

- **Longitudinal Controlling:** When to throttle? When to brake?
- **Lateral Controlling:** When to steer? How much to steer?
- **State Estimation & Localization:** Where am I?
- **Computer Vision & Scene Understanding:** Where can I drive? Where are the lanes? Where is the sidewalk? Is the car in the opposite lane approaching dangerously?

Inevitably, the most powerful answers to such questions are hidden under most enterprises' intellectual property or top-notch conferences.

This thesis focuses on exploring solutions to the problems as mentioned earlier, which are related to vehicle control automation, state estimation, and localization, along with visual perception for road scene understanding.

In more detail, this current exploitation contributes to the following:

- We begin by generating samples from CARLA's highly realistic environment. Such patterns include geographic information from LIDAR and GPS sensors, along with speed and orientation from the Inertial Measurement Unit of the car. For the visual perception scenarios, screenshots from CARLA's camera are used, along with information from the depth and segmentation neural network API that CARLA provides.

- In the automation scenario, we implement a PID controller for the throttling and braking operations, having as measured process variable the velocity of the car for each time frame. The Stanley controller is then introduced for the steering action of the vehicle, which takes positions of the known map and creates the ideal journey path for the car. We then compare it to the ground truth trajectory.
- Next, having generated data through a test drive, we try to determine where the car is in the world frame by localizing it, which is essential for automated controlling. Explicitly, we define an accelerated motion model, using data from the IMU, and fuse them with the position estimates from the GPS and LIDAR sensors, by applying the Extended Kalman Filter technique. We also visualize the effects of sensors dropping out to showcase the importance of sensor fusion in the task of a vehicle's localization.
- A remarkable part of this thesis is related to the visual perception of self-driving cars. At first, we extend the aforementioned state estimation scenario and explore how camera information can be used to localize in the vehicle frame, accomplishing visual odometry with stereo cameras. In the latter part, we leverage CARLA's segmentation neural network and combine its information to estimate lanes and drivable surfaces in 3D, using popular computer vision algorithms. Last but not least, we exploit the 3D information from the car's camera and merge it with object detection, filtering out unreliable results in order to showcase how collision safety can be attained.

1.3 Thesis Structure

The rest of this dissertation is organized as follows:

Chapter 2 provides a showcase of the CARLA environment, its segmentation deep neural network, and the generation of the datasets.

Chapter 3 describes the methodology for the PID and Stanley controller in the task of automating the vehicle's motion, both longitudinally and laterally.

Chapter 4 explores Extended Kalman Filtering, an excellent technique for sensor fusion. In our case, the data for the motion model is provided through CARLA's IMU while the measurement positions are given from its LIDAR and GPS sensors.

Chapter 5 focuses on processing camera data for various tasks. First, visual odometry for localization is achieved by estimating the camera motion between frames. Next, we present various computer vision approaches in combination with CARLA's segmentation neural network to emphasize road scene understanding for road and lanes estimation, along with the development of a collision system in order to avoid danger impact.

Chapter 6 summarizes the thesis and concludes it.

Chapter 2

CARLA: An Open Simulator for Autonomous Driving Research

2.1 About

Research in autonomous urban driving is blocked by the costs of infrastructure and many logistical difficulties in the real, physical world. Developing and maintaining even one driverless robot car requires notable supplies and human resources. Besides, a single vehicle is insufficient for collecting the required data that cover the plenitude of corner cases that must be treated for development [3].

FIGURE 2.1: CARLA's simulation environment [3]

.

CARLA *(Car Learning to Act)* is an open simulator for urban driving. CARLA is developed to encourage development of self-governing civil driving systems. The simulation platform supports flexible setup of sensor suites and provides data that can be used to design and develop driving strategies, such as **GPS coordinates, speed, acceleration, and detailed information on infractions or collisions**. Also,

a wide range of conditions about the environment can be specified, including time of the day, weather, and more [3].

CARLA is written in C++ code and rendered in Unreal Engine, which outputs a high-resolution, realistic simulation. It supports code scripts integration through its Python Client-Server API, which we are going to use extensively for the experiments of this thesis.

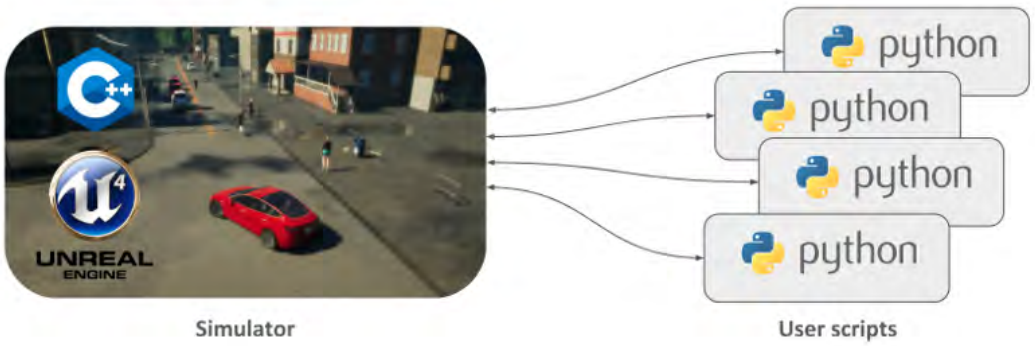


FIGURE 2.2: CARLA's environment and Client-Server Example [3]

.

2.2 Datasets

CARLA can be exploited to generate the datasets for each experiment. CARLA supports a recording script *(recording.py)* which logs all individual measurements, and anyone could modify it to their needs to generate specific data types. Also, it provides functions so we can output exclusive screenshots from the stereo cameras through its API. The documentation to *Connecting a Python client* is outside the scope of this thesis; although, it can be found on CARLA's website [3].

For the automated control simulation, we have a dataset with 3 features. The first two attributes display the ground truth X, Y position of a car driving in a racetrack map, and they are generated through a 'test' drive. The third cell is the reference velocity of the vehicle. The map of the current world in CARLA's simulation environment can be retrieved by $map = world.get\_map()$ while the waypoints can be retrieved by $waypoint = map.get\_waypoint(vehicle.get\_location())$. The reference velocity is arbitrarily edited.

$$\begin{vmatrix} x_1 & y_1 & v_{f1} \\ x_2 & y_2 & v_{f2} \\ .. & .. & .. \\ x_N & y_N & v_{fN} \end{vmatrix} \tag{2.1}$$

For the localization scenario, we use data from the Inertial Measurement Unit of CARLA's sensor which contains a *StampedData* object with the IMU specific force and rotational velocity data in each time frame as long as data from the GNSS and LIDAR sensors which contain the X, Y, Z position measurement for each time frame it's available. Once again, the Python API from CARLA is exploited to get the information of the actor (vehicle). Example commands include *actor.get_location()* and *actor.get_acceleration()*.

For the visual perception scenarios, frames of CARLA's camera sensor are exported. For the visual odometry task, we use fifty-two subsequent screenshots. For the road scene understanding, we use three image frames where other vehicles or pedestrians are in sight. The objects are denoted as *sensor.Image* objects. It is noted that each frame or screenshot is synchronized with its depth map from CARLA's Camera depth map output and CARLA's Semantic Segmentation output by using CARLA's *background.save()* command.

For the record, other datasets are becoming available as time passes by, meaning that they could be used and contribute to the literature, like the KITTI Vision Benchmark Suite or comma.ai's comma2k19 dataset which provides over 33 hours of commute in California's 280 highway.

2.3 Built-in Semantic Segmentation Neural Network

CARLA's "Semantic Segmentation" camera classifies every object in the view by displaying it in a different color according to the object class. For example, pedestrians appear in a different color than vehicles or sidewalks. Actually, it is a convolutional neural network that can do multiclass classification and provide its outputs through the API, thus creating the semantic segmentation camera sensor which produces *carla.Image* objects.

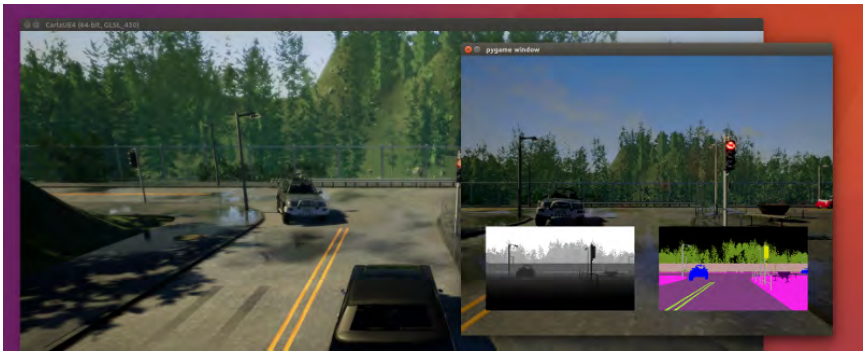


FIGURE 2.3: CARLA Simulation Environment along with Semantic Segmentation and depth cameras [3]

.

The perception stack of CARLA is built upon a semantic segmentation network based on RefineNet [10]. The network is trained to classify each pixel in the image into one of the following semantic categories:

C = *road, sidewalk, lane marking, dynamic object, miscellaneous static* [3]

CARLA's highly citated paper states that their network is trained on 2,500 labeled images produced in the training environment using CARLA. The output of their segmentation network is used to compute an obstruction mask that aims to include pedestrians, vehicles, and other hazards [3]. Also, CARLA's semantic segmentation neural network can estimate the likelihood of being at an intersection by using a binary scene classifier (intersection/no intersection) [7]. The neural network of CARLA has been trained on 500 images for the two classes of intersection and no intersection.

Chapter 3

An Automated Control System

3.1 Longitudinal Vehicle Control: PID Controller

A proportional–integral–derivative controller or, in short, PID controller, is a control loop feedback mechanism. These controllers are widely used in the industry of control systems. A PID controller calculates an error value $e(t)$ as the difference between a desired variable (denoted setpoint SP) and a measured actual variable (denoted process variable PV). Such PID controllers attempt to correct the actual variable to its desired one based on its proportional, integral, and derivative terms (denoted P, I, and D).

Practically, it is an automated way to correct a function in the control system responsively. We are going to use such a PID controller later in our experiment in order to automate the throttling and braking operations for the autonomous vehicle. The cruise control on an everyday car could form an example. For example, ascending a hill would lower speed if only the engine power applied was constant. In reality, this does not happen due to the controller's PID algorithm, which restores the measured speed to the desired speed with almost minimal delay [29] .

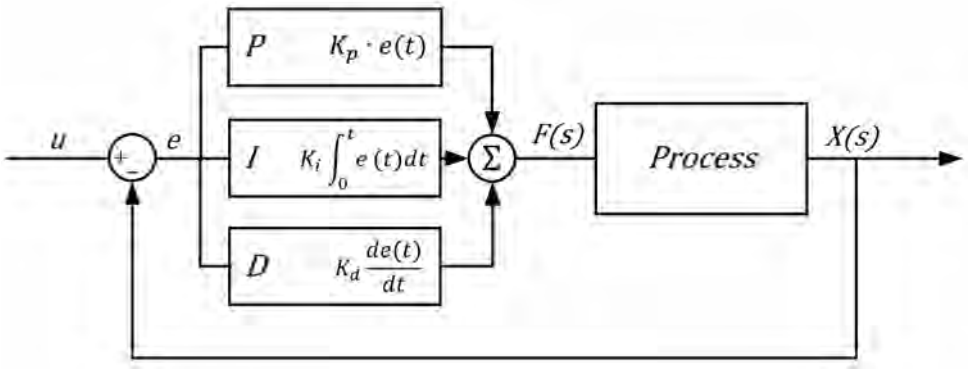


FIGURE 3.1: Overview of a PID Controller, where the process variable is denoted as u

3.1.1 Proportional Response

The proportional component of the controller is the difference between the set point variable (SP) and the process variable (PV). This difference can be defined as the error term. *The proportional gain (Kp) determines the ratio of output response to the error signal. In general, increasing the proportional gain will increase the speed of the control mechanism response.* Nonetheless, if the gain of the proportional component is really large, the process variable will start oscillating[15]. If K_p gets increased more and more, the oscillations then will become even larger and the system's signal will become unstable and may even oscillate out of control.

$$\text{Proportional Response} = K_p * e(t) \tag{3.1}$$

3.1.2 Integral Response

The integral component I gives a correction proportional to the integral of the error and its goal is to reduce the tracking error near to zero. The Integral Gain is denoted as K_i [15].

$$\text{Integral Response} = K_i * \int_0^t e(t)dt \tag{3.2}$$

3.1.3 Derivative Response

The derivative component causes the output to decrease if the PV is increasing quick enough. The response of the derivative component is essentially proportional to the rate of change of the actual variable. The gain to the derivative response is denoted as K_d. Increasing the derivative time T_d parameter will increase the control system response speed. Most practical control systems in the industry use minimal *derivative time (T_d) because the Derivative Response is highly sensitive to noise in the process variable signal. If the sensor feedback signal is noisy or if the control loop rate is too slow, the derivative response could even make the control system unstable* [29].

$$\text{Derivative Response} = K_d * \frac{de(t)}{dt} \tag{3.3}$$

3.1.4 Tuning

The operation of setting the PID gains to optimal values for a control system is called tuning. Most of the times, the gains can be defined by a trial and error method or

others like the Zeigler-Nichols one, which heuristically sets the integral and derivative gains to zero to find the ultimate gain for K_p. Control System Designers may sacrifice one characteristic of their control loop for another, according to their needs.

The following figure describes how tuning K_p, K_i, K_d affects the closed loop response or else, the control system:

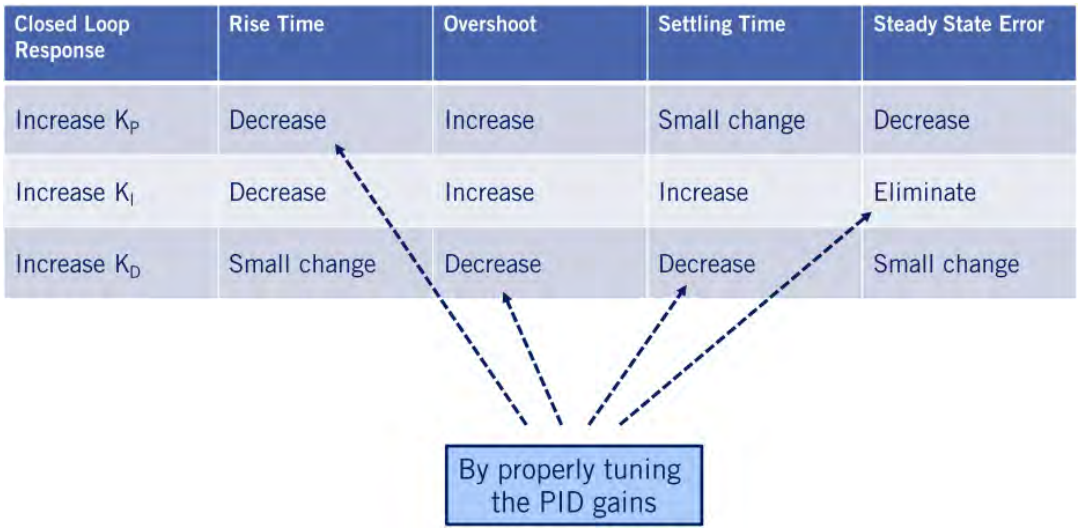


FIGURE 3.2: PID Tuning effects [6].

3.2 Lateral Vehicle Control: Stanley Controller

The problem of steering is equal to the one of having the car to follow the desired path. Lateral Controllers are responsible for creating paths or estimating trajectories. Generally, they define an error relative to the desired path, which they try to minimize to zero while satisfying the input constraints. Also, they may have additional dynamic considerations to balance forces that are applied from the environment to the car.

In this section, we are going to introduce a full geometric controller, the Stanley Controller. It is the path tracking approach that Stanford University's team designed for their automobile in the DARPA Grand Challenge [21].

3.2.1 Heading error

The term heading error defines the angle measurement by which the vehicle should steer, and it is relative to the desired trajectory. To define the angle, a vehicle reference point is needed. Stanley uses the front axle's center as the reference point [21].

The desired heading of a vehicle in physics is defined as $\delta(t)$, when the heading relative to the trajectory is defined as $\psi(t)$, where t is the time frame.

3.2.2 Cross-track error

Cross-track error, noted by e, is the distance from the vehicle reference frame to the closest point on the reference path.

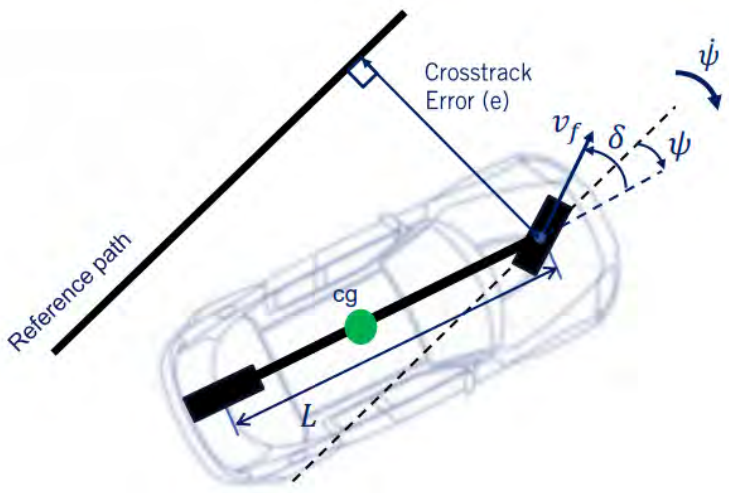


FIGURE 3.3: The cross-track's error relationship to the reference path, the heading error, and the current pose [27]. The heading error is noted by ψ.

The rate of the cross-track error's change is calculated as:

$$e(t) = v_f(t) * sin(\psi(t) - \delta(t)) \tag{3.4}$$

3.2.3 Stanley Control Law

Stanley's approach to the heading control has three requirements combined. The first requirement is to steer to align the vehicle heading with the desired heading, which is proportional to the heading error.

$$\delta(t) = \psi(t) \tag{3.5}$$

Then, the goal is to steer in order to eliminate the cross-track error:

$$\delta(t) = tan^{-1}\frac{k * e(t)}{v_f(t)} \tag{3.6}$$

The gain k is determined experimentally. The formula is substantially proportional to the error $e(t)$ and inversely proportional to the speed of the vehicle $v_f(t)$. The use of the inverse tangent is to limit the effect for significant errors, as their paper states [21].

Of course, a vehicle has limitations on its steering angles. For example, it can not steer by 180 degrees. That's why Stanford's team consider minimum and maximum steering angles $[\delta_{min}, \delta_{max}]$.

By combining the three above requirements, the Stanley Control Law finally formulates as:

$$\delta(t) = \psi(t) + tan^{-1}\frac{k * e(t)}{v_f(t)} \tag{3.7}$$

where

$$\delta(t) \in [\delta_{min}, \delta_{max}]. \tag{3.8}$$

3.3 Experiment #1: Automating a vehicle

So, for our experiment, we use a dataset containing an X, Y, position, and speed for an UnrealEngine map in the simulator. These are generated through a test drive in CARLA, as explained in Chapter 3. The map that is used is the *RaceTrack* map in CARLA, which contains no third objects so the car can not collide with anything.

3.3.1 Longitudinal Control with a PID Controller

The control loop system that we designed for the CARLA's vehicle contains a PID controller, concerning the longitudinal control of. Its flowchart is presented below, in Fig. 3.4.

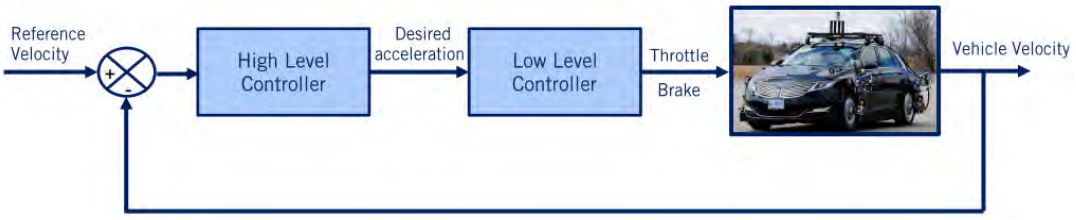


FIGURE 3.4: Flowchart of a PID controller for a self-driving car [27]

.

In our demonstration the low-level controller is fully controlled by CARLA, meaning that we only have to design the PID (high level) controller. CARLA handles everything else.

The parameters used contain the desired or reference speed v_d, the vehicle actual speed v and the acceleration input u.

The PID controller is equal to the following:

$$u = K_p * (v_d - v) + K_i * \int_0^t (v_d - v)dt + K_d * \frac{d(v_d - v)}{dt} \tag{3.9}$$

The gain parameters for the PID Controller are the K_p, K_I, and the K_D variables. After many iterations of trial-and-error experimentation, we set them equal to 1, 0.1, and 0.1 accordingly.

The CARLA API has two variables that define the longitudinal control of the car, the Throttle position T_p and the Brake position B_p.

If the acceleration input is equal or greater than zero, then we set the Brake Position as 0 and the Throttle position as u.

While if it is negative, then we only want the car to brake, meaning that we set the brake position equal to $-u$.

Algorithm 1: Connecting the PID controller to CARLA

Result: Returns estimated velocity based on the PID Controller results

1 **for** *each time frame* **do**
2 | If u>= 0: $T_p = u, B_p = 0$
3 | If u< 0: $T_p = 0, B_p = -u$
4 **end**

3.3.2 Lateral Control with the Stanley Controller

For the lateral control, we first calculate the difference between the pose of the vehicle and the desired pose (in radians). The actual and the desired orientation or angles of the vehicle can be calculated as $\theta = atan2(y, x)$ with the NumPy package [16]. The $atan2(y, x)$ command calculates the element-wise arc tangent of y/x choosing the quadrant correctly. It is used to define the heading error:

$$\psi(t) = \text{path line angle} - \text{actual angle} \tag{3.10}$$

For the next step, the crosstrack error is calculated :

$$e(t) = min(dist(car_{xy}, waypoints)^2) \tag{3.11}$$

where *dist* is the Euclidean distance between the points. The car's xy positions are provided for each time frame by the CARLA simulation environment while the original waypoints are already known from the start.

Having the cross-track error e, we can then calculate the cross track steering formula explained above and provide the following steering input to CARLA:

$$\delta(t) = \psi(t) + tan^{-1}\frac{k * e(t)}{v_f(t)} \tag{3.12}$$

keeping

$$\delta(t) \in [-1.22, 1.22]$$

as CARLA API's steering limits in radians and k equal to 0.1, as presented in the original paper from Stanford's team [21].

3.3.3 Implementation & Evaluation

The above techniques were implemented in Python, using the NumPy package in a Jupyter Notebook environment and inputs extensively from the CARLA API. As mentioned in Chapter 2, communication to the CARLA simulation environment was implemented through the Client-Server API and this mechanism was exploited for the whole dataset generation.

In the following snippet, we will showcase the main algorithm inside a CARLA environment.

```
# PID Controller
dt = t - self.vars.previous_time
error = np.abs(v_desired - v)

proportional = self.vars.kp * error
integral = self.vars.ki* self.vars.integral_value * error * dt
derivative = self.vars.kd * (error - self.vars.last_error/ dt)

vehicle_velocity = proportional + integral + derivative

if vehicle_velocity < 0:
    vehicle_velocity = 0.0
    brake_output = - vehicle_velocity
else:
        if (vehicle_velocity > 1) :
        vehicle_velocity = 1
    brake_output = 0

throttle_output = vehicle_velocity
```

```
# Stanley Controller
```

```
# 1. Calculate heading error psi(t)
yaw_path = np.arctan2(waypoints[-1][1]-waypoints[0][1], waypoints[-1][0]-
    waypoints[0][0])
heading_error = yaw_path - yaw #

# 2. Calculate crosstrack error and the difference of rate as in theory
current_xy = np.array([x, y])
crosstrack_error = np.min(np.sum((current_xy - np.array(waypoints)[:, :2])
    **2, axis=1)) # Euclidean Distance

crosstrack_error_difference = np.arctan(k_e * crosstrack_error / (k_v + v)
    )

# 3. Implement the final math type
steer_expect = heading_error + crosstrack_error_difference

steer_expect = min(1.22, steer_expect) # CARLA limits to [-1.22, 1.22]
    radians
steer_expect = max(-1.22, steer_expect)
```

The implementation of the PID and the Stanley Controller resulted in the following trajectories that were printed with Python Code through with the *matplotlib* package.

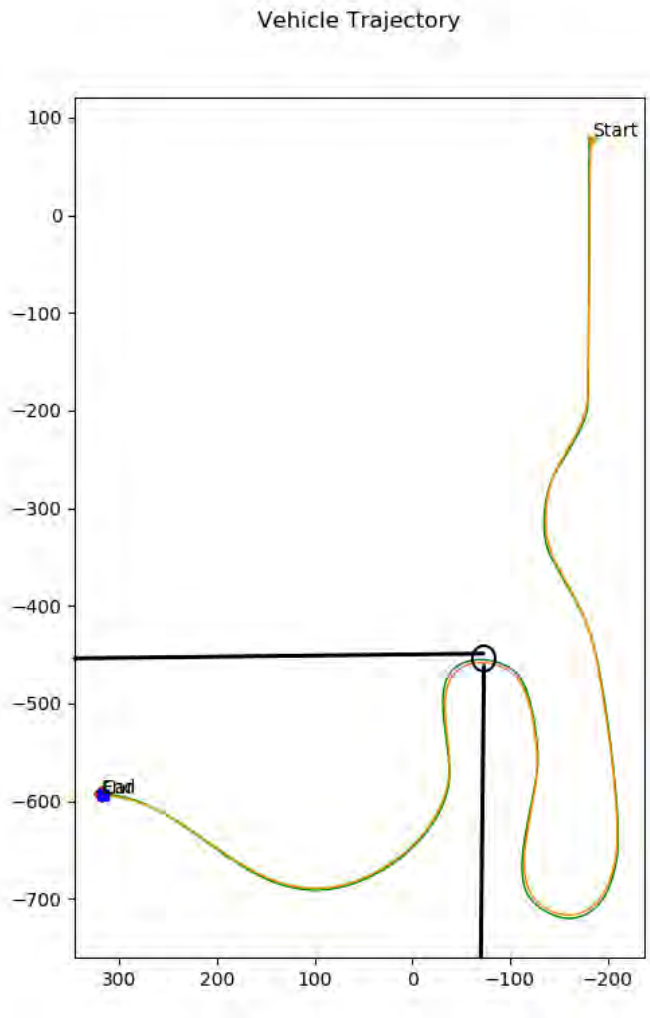


FIGURE 3.5: Generated trajectory

We can observe the ground original X, Y positions in blue while the automated vehicle followed the trajectory of the green line. It succeeded following the desired path by 91.2% without any concerning bypass. Of course, there are minor differences visually apparent when the vehicle is needed to steer in a high angle, like at point [-90, -450]; although, they do not look worrying as the estimated path (or signal) does not oscillate and follows the desired one reliably.

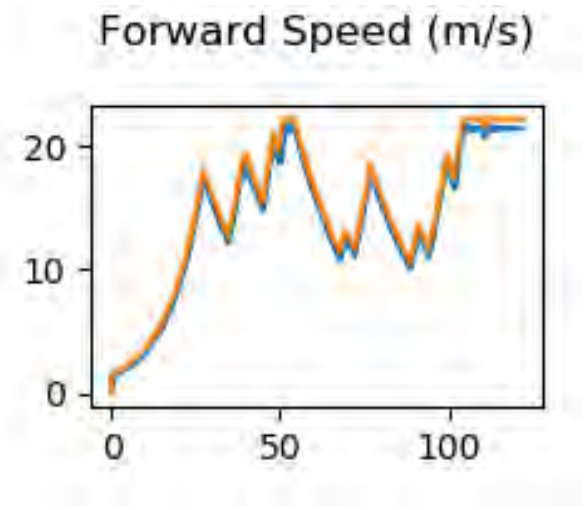


FIGURE 3.6: The Y-axis determines the controlled forward speed in m/s while CARLA's timestamps compose the X-axis

Last but not least, the longitudinal control that we applied with the simple PID controller produces satisfying results when attempting to follow the desired speed. . In the blue lines, we can see the ground truth reference speed, while the actual speed that the control loop system calculates is in orange having a mean absolute error of $0.23 m/s$, showing the power of the PID Controller.

Chapter 4

State Estimation & Localization

In this chapter, we are going to introduce how different sensors can be combined and fused with a technique called *Kalman Filtering* [28] in order to localize a self-driving car successfully. Sensors arecan be times fuzzy, inaccurate or even damaged, thus, we can not rely only on one sensor for essential tasks like localization, where safety is a top requirement. Kalman Filtering is a popular technique for sensor fusion, and it was even used in the Apollo program for the same purpose of state estimation and localization. Its inventor is Rudolf E. Kálmán, who was also awarded the National Medal Of Science in 2009.

4.1 Sensors

Sensors are the components that collect information, which we can process in order to make a vehicle artificially intelligent. More formally, a sensor is a device that measures or detects a property of the environment or changes to a property. They are categorized into two categories, the *exteroceptive* which is used for the surroundings, and the *proprioceptive*, which is used for the internal parts of the vehicle [9]. In the following subsections, we explain briefly the sensors that are used in our experiments. This can include events like trajectory estimation, localization, object detection, semantic segmentation, object localization, and more.

4.1.1 Camera

A camera is essential for correctly perceiving the environment of a vehicle. Its key components are the resolution, the field of view, and the dynamic range of it. Specifically, stereo cameras are the ones that are most needed because they enable depth estimation from image data, something essential for object localization tasks and not only. Last but not least, cameras are inexpensive, and they can provide much information about our surroundings. The camera and its parameters are explained in a more detailed way in Chapter 5.

4.1.2 LIDAR

LIDAR (Light Detection and Ranging) is the combination of light and radar. A LIDAR sensor can measure the distance to a target by illuminating it with laser light. The pulses that get reflected then get measured, and they can be used in order to represent maps. Sometimes it is also called 3D laser scanning. Many parameters define a LIDAR sensor like the number of beams, the points per second, the rotation rate, and the field of view. LIDAR results are pretty accurate, but they are hard-to-process sensors and most notably, pretty expensive [9].

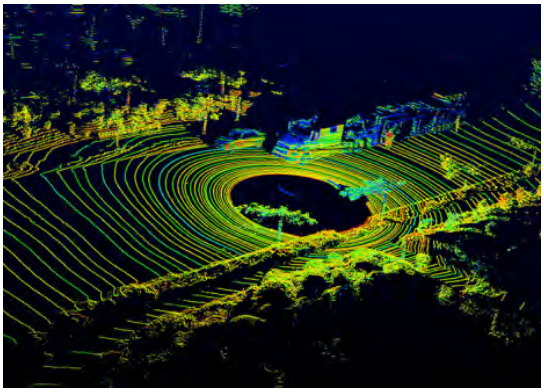


FIGURE 4.1: LIDAR [13]

4.1.3 Global Navigation Satellite Systems and Inertial Measurement Units

GNSS and IMU provide a direct measure of "ego vehicle" states like:

- position, velocity (GNSS)
- angular rotation rate (IMU)
- acceleration (IMU)
- heading (IMU, GPS)

FIGURE 4.2: Satellites are the components used for GNSS localization. [13]

Note that Global Navigation Satellite Systems have different accuracies like RTK, PPP, DGPS. We use CARLA'S Global Positioning System (GPS).

4.2 Linear Kalman Filter

Before explaining the Extended Kalman Filter [28] that we used for the localization problem, a description of the simple Linear Kalman Filter is needed first. The (linear) Kalman Filter is based on estimating the state or position of a system based on its motion model (e.g., speed, gravity) and then correcting this estimation based on a measurement model (e.g., GPS, visual odometry and more). It is modeled on a Markov chain, and it covers errors that may include Gaussian Noise, thus making it a probabilistic technique for sensor fusion.

The modeling of the Kalman Filter requires some modeling of the process. In order to model the Kalman Filtering process, we specify the following matrices [28]:

- F_k, the state-transition model;
- H_k, the observation model;
- Q_k, the covariance of the process noise;
- R_k, the covariance of the observation noise;
- and sometimes B_k for each time frame k, which is the control model for the input, as following.

The Kalman filter considers a vehicle's true state at time x_k is derived from the state at x_{k-1} according to:

$$\mathbf{x}_k = \mathbf{F}_k(\mathbf{x}_{k-1}) + \mathbf{B}_k(\mathbf{u}_k) + \mathbf{w}_k \tag{4.1}$$

where

- F_k is the state transition model which is applied to the previous state x_{k-1};
- B_k is the control model for the the control vector u_k;
- w_k is the process noise which is assumed to be drawn from a zero mean multivariate normal distribution $\mathcal{N}$, with covariance Q_k : $w_k \sim \mathcal{N}(0, Q_k)$

At the time k an observation (or measurement) z_k of the actual state x_k is made according to

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \tag{4.2}$$

where

- H_k is the observation model which is a correspondence between the true space and the observed one
- v_k is the observation noise which is assumed to be zero mean Gaussian white noise with covariance $R_k : w_k \sim \mathcal{N}(0, Q_k)$

The algorithm could be visualized as:

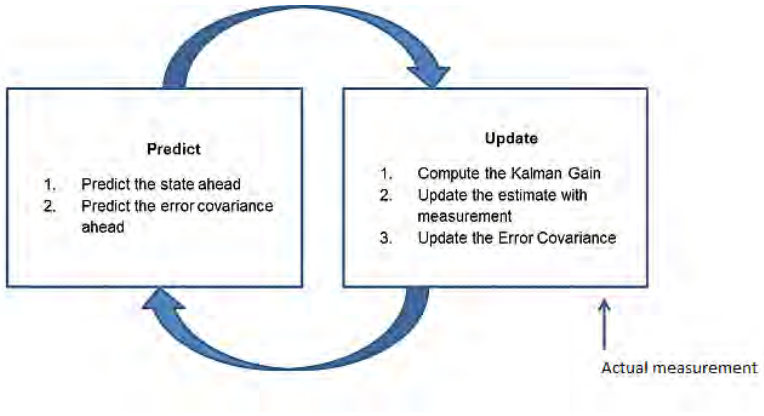


FIGURE 4.3: Flow chart of the Iterative process [22]

4.3 Extended Kalman Filter

Inevitably, Kalman Filtering works only for linear models. The majority of real-life models are not linear, which expresses the need for a non-linear Kalman Filter, the Extended Kalman Filter (EKF)[28].

The EKF (Extended Kalman Filter) can use non-linear state transition and observation models. The functions that are used for these models are of a differentiable type. [28]

$$\begin{aligned} \mathbf{x}_k &= f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k \\ \mathbf{z}_k &= h(\mathbf{x}_k) + \mathbf{v}_k \end{aligned} \tag{4.3}$$

The function f uses the previous estimation to compute and predict a state. Also, the function h uses the predicted state in order to calculate and predict a measurement. However, the f and h matrices can not be utilized to the covariance directly; a Jacobian matrix of partial derivatives is needed and therefore computed [28].

In each timeframe, the Jacobian matrix is evaluated with current predicted states. This process substantially provides a linear estimation of the non-linear function around a current point, by computing a linear approximation using a first-order Taylor expansion on the operating point[28].

The complete algorithm could be visualized as:

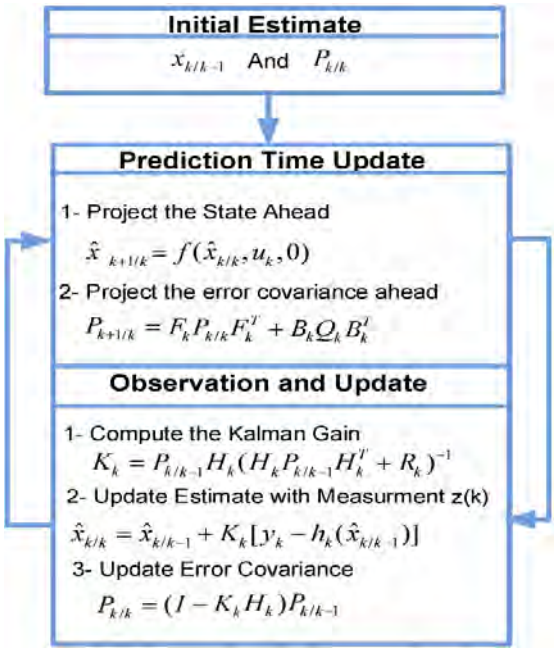


FIGURE 4.4: Overview of the Extended Kalman Filter Process as stated in ResearchGate.

4.4 Experiment #2: Sensor Fusion and Localization

For the localization experiment, the Inertial Measurement Unit of the CARLA's simulation environment smart car is leveraged for the motion model. Also, GNSS and LIDAR measurements are exploited for the correction or measurement model. The data has been recorded with CARLA's API, as mentioned in Chapter 2.

The motion model input consists of specific force f and rotational rate ω_k from our IMU [3]:

$$u_k = \begin{bmatrix} f_k \\ \omega_k \end{bmatrix} \in R^6 \tag{4.4}$$

The measurement model input consists of the GNSS and LIDAR measurements that have the following format [3]:

$$y_k = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \in R^3 \tag{4.5}$$

Of course, in order to formulate the XYZ position of a car, its position, orientation, and its velocity for the motion model are needed. The accelerated motion of a vehicle is defined as:

$$\text{Position: } p_k - p_{k-1} = v_{k-1}\Delta t + a\frac{\Delta t^2}{2} \tag{4.6}$$

$$\text{Velocity: } v_k - v_{k-1} = a\Delta t \tag{4.7}$$

$$\text{Orientation: } q_k = \omega_{k-1} * \Delta t * q_{k-1} \tag{4.8}$$

where $a = f_{k-1}$, the force rate from the CARLA's inertial measurement unit.

Thus, the Jacobian Matrix F_{k-1} is defined as:

$$F_{k-1} = \begin{bmatrix} I_3 & I_3\Delta t & 0 \\ 0 & I_3 & -f_{k-1}\Delta t \\ 0 & 0 & I_3 \end{bmatrix} \epsilon R^{9x9} \tag{4.9}$$

where I_3 is the 3x3 identity matrix.

Also, the H_k vector is denoted as $H_k = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$ since the measurement updates refer only to the first component, the position of the vehicle and not to the velocity or the orientation.

Of course, we suspect zero mean and covariance R_{GNSS}, R_{LIDAR} Gaussian noise signals, combined in a diagonal $2x2$ matrix, for the measurement model update. The same applies to the specific force and the rotational rate that come from CARLA's inertial measurement unit.

A high-level flowchart of the proposed methodology that exploits the Extended Kalman Filter fusion is below, while the reader can get more comfortable in this advanced methodology by reading the interactive book *Kalman and Bayesian Filters in Python* [8], which provides starter codes and extensive explanations.

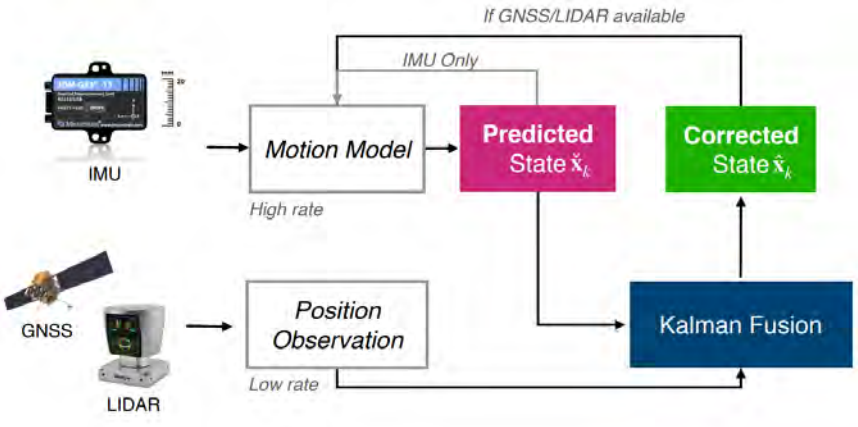


FIGURE 4.5: High-level overview of the sensor fusion experiment [26]

Furthermore, after defining the above problem definitions in Python with a signal processing library such as *filterpy* [8], the Extended Kalman Filter can return outputs of our positions estimates, for each time frame, which follow the Euler Angles format:

$$p_{est} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \in R^3$$

We can then visualize with *matplotlib* each time frame and the three axes.

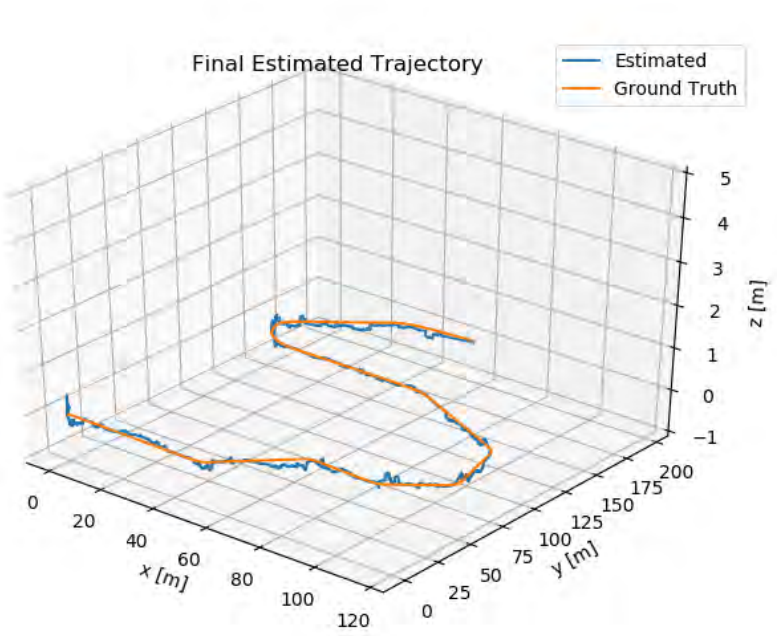


FIGURE 4.6: Comparison of the estimated trajectory to the ground truth one

The localization estimation is a bit jaggy on the Z-axis but other than that, it is close to the ground truth and does not suffer by any big misprediction, having a MAE of $0.32m$.

In addition, it is interesting to observe what happens if we remove the GNSS and the LIDAR data integration and plot only the position outputs from IMU.

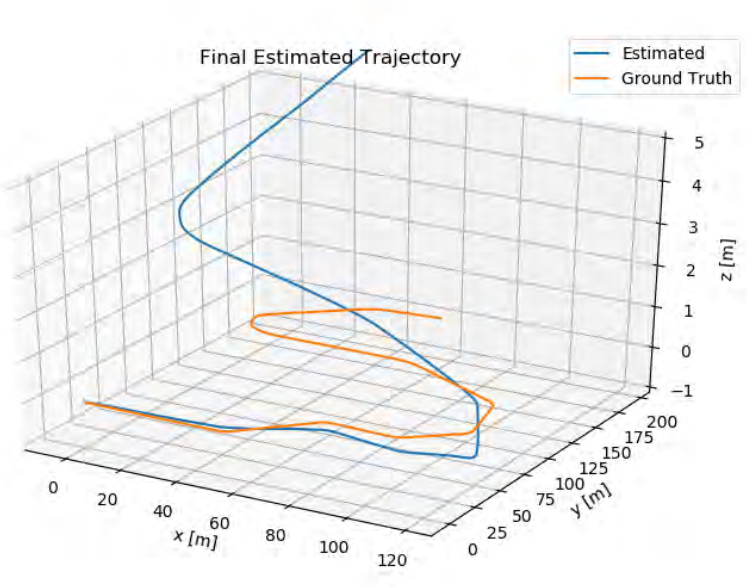


FIGURE 4.7: The effect of dropping out the measurement sensors and relying only on the motion model.

The above plot shows and verifies that we can not only rely on one sensor for such a task since even by defining models mathematically correctly; sensors some times may be biased or incorrect and can not produce a correct result, especially when concerning that in a significant time scale. Sensor fusion is needed, and it is used widely in every localization application in the modern days.

Chapter 5

Visual Perception

In this Chapter, we are going to introduce how computer vision is implemented in self-driving cars and contribute to the visual perception of the environment. Two extensive examples will be demonstrated. First, we will show how the information from a camera can be used in order to localize the car in the camera reference frame, extending the previous chapter. In the latter part, we are going to process outputs from CARLA's segmentation neural network and combine them with classical computer vision techniques in order to achieve successful results on plane fitting, lanes estimation and avoiding collision on a life-like environment.

5.1 3D Computer Vision

Before digging into experiments, we need to define the basics of 3D Computer Vision. Computer vision tasks often exploit analyzing and processing information from a camera. Most modern-day cameras follow the pinhole camera model [2]:

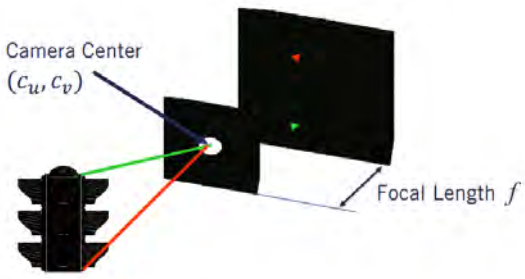


FIGURE 5.1: The pinhole camera model.

As explained in Chapter 3, the camera sensor is a critical sensor due to the vast amount of information it can capture, while its price is low nowadays.

5.1.1 Reference Frames

An image from a camera is 2D (u, v), while the world is expressed in 3D coordinates (X_w, Y_w, Z_w). In order to translate the projective geometry of an image in 3D world coordinates, we need to translate it to camera coordinate systems (X_c, Y_c, Z_c). In total, this sums up to 3 different coordinate systems, the world system, the camera system, and the image system.

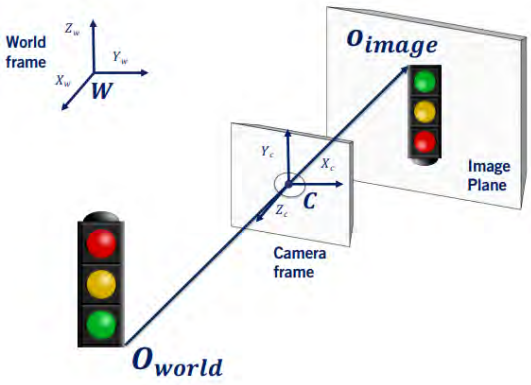


FIGURE 5.2: The world, camera and image reference frames.

In order to project world coordinates to camera coordinates, a transformation multiplying with the camera's extrinsic parameters is needed.

$$o_{camera} = [R|t]o_{world} \tag{5.1}$$

$$o_{world} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \in R^3 \tag{5.2}$$

The rotation matrix R and the translation matrix t are covered in the following subsection.

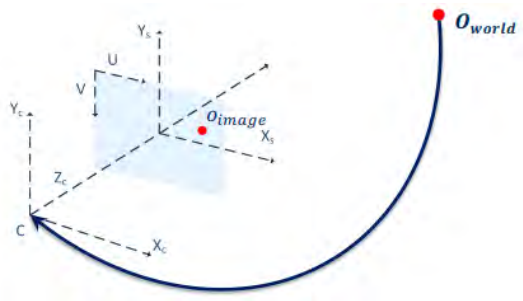


FIGURE 5.3: Transformation from the world frame to the camera frame.

In order to project camera coordinates to image coordinates, a different transformation is needed, which is based on the intrinsic parameters of the camera that we defined above, the focal length and the center of the camera:

$$o_{image} = K * o_{camera} = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} o_{camera} \tag{5.3}$$

The K matrix is called the intrinsic matrix and may sometimes include a scale s also for scaling operations.

So, finally, a projection from world coordinates to image coordinates would look like this:

$$o_{image} = P * o_{world} = K[R|t]o_{world} \tag{5.4}$$

using homogeneous formats in the world frame $\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$ for valid linear multiplications.

Last but not least, image coordinates can be transformed to pixel coordinates with a simple multiplication:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{z} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{5.5}$$

5.1.2 Stereo Cameras and Depth Perception

Depth is one key element in order to perceive the environment, but inevitably, it is hard to compute it with only one camera sensor. Instead, stereo cameras are used. A stereoscopic sensor is constructed from two identical cameras which have parallel optical axes.

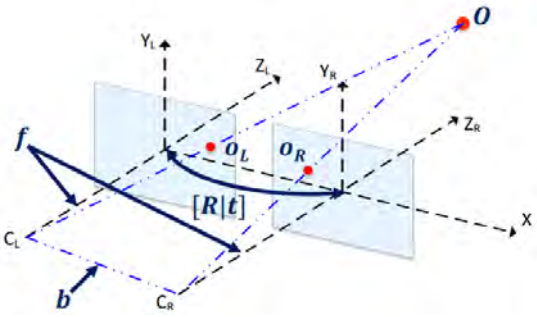


FIGURE 5.4: An example stereo camera model [2]

Depth is calculated through the disparity of the differences x_L and x_R that are depicted in the following figure.

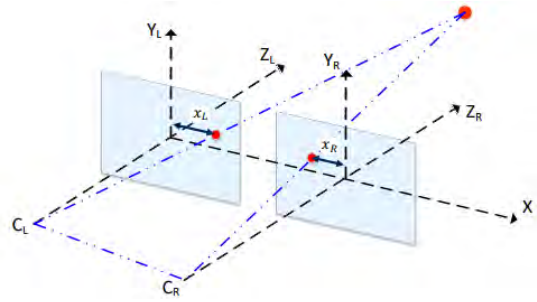


FIGURE 5.5: Example of calculating the disparity and the depth from two identical cameras.

Luckily, CARLA has a depth sensor provided for us, which is essentially computing all the fuzzy maths within two identical cameras, providing for us the depth metric in an image between 0 and 1000 meters.

5.1.3 Transformations

Let's consider a point with coordinates p_1 in one camera coordinate system and the same point with coordinates p_2 in the second camera coordinate system.

The relation between them is:

$$p_2 = Rp_1 + t = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} p_1 \tag{5.6}$$

where R and t are the rotation and translation matrices. A rotation matrix rotates points about an axis:

$$P' = RP \tag{5.7}$$

An example of a rotation matrix multiplication in 2D would be:

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} \tag{5.8}$$

A translation matrix is a shift of the origin point to the new point in 3 dimensions:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} \delta x \\ \delta y \\ \delta z \end{bmatrix} \tag{5.9}$$

The same mindset applies in 3D matrices and requires more complex computations, which can be computed easily with the use of the OpenCV package, an open-source library for Computer Vision tasks [1].

5.2 Image Features, Detectors & Descriptors

Features in an image are the points of interest in the image. Such points of interest should have the following characteristics: saliency, repeatability, locality, quantity, efficiency.

Some pixels in an image may contain more information than others. Such are patches with significant contrast changes (edges) or gradients in at two different orientation (corners). Many algorithms can do the task of feature detection, such as the Harris Corner Detection, Harris-Laplace, LOG, DOG detector, and more.

Here is an example of performing the Harris Corner Detector in a real-life car camera screenshot.

FIGURE 5.6: Example of an applied Harris Corner Detector in CITYSCAPES [24]

Empirical validation is always needed in order to choose the best extractor based on the application that we perform it on.

Sometimes, image features may not be enough. Let us consider a data leakage error where some pixels such as there points of interests get removed. Alternatively, a

transformation in the image could happen after some processing. In any way, there is a need for a better description of an image, and that is where feature descriptors are used. Instead of characterizing only the $[u, v]$ position of an *interesting* pixel, they also provide an N-dimensional vector with the summary of the image information around the detected feature. Such feature descriptors must be robust and invariant to the translation or rotation of the images and any other transformation like scaling or illumination changes. Also, they must be distinctive in order not to mess them up.

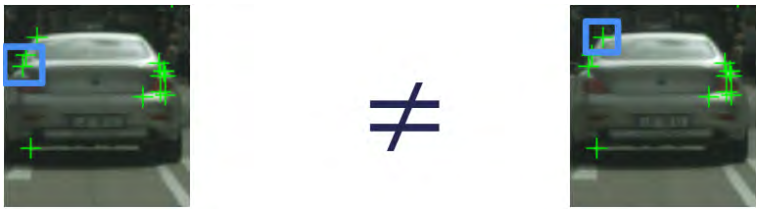


FIGURE 5.7: A feature descriptor applied in CITYSCAPES [24]

5.2.1 Scale Invariant Feature Transform (SIFT) descriptors

SIFT descriptors can be used in order to detect keypoints in images, even on different scales. The Difference of Gaussian, called DoG, is calculated. This difference consists of the Gaussian blurring difference of an image. 'DoG' essentially is a detector for blobs, considering various sizes due to the change in σ, the returning and scaling parameter. A Gaussian kernel with low *sigma* gives high value for a small corner while it returns a high *sigma* for a larger corner. The above means that we can find the maximum local values in scale and space which gives us a list of (x, y, σ) values which means there is a potential keypoint at (x, y) at σ scale. This process is done for different octaves of the image in the Gaussian Pyramid [11].

Lastly, a keypoint descriptor is created, and a 16x16 neighborhood around the key point is necessitated. It is separated into 16 sub-blocks of 4x4 size. For each sub-block, an eight bin orientation histogram is created. So a total of 128 bin values are available, each one represented as a vector to form a key point descriptor. In addition, several measures are taken to achieve robustness against illumination changes, rotation [19].

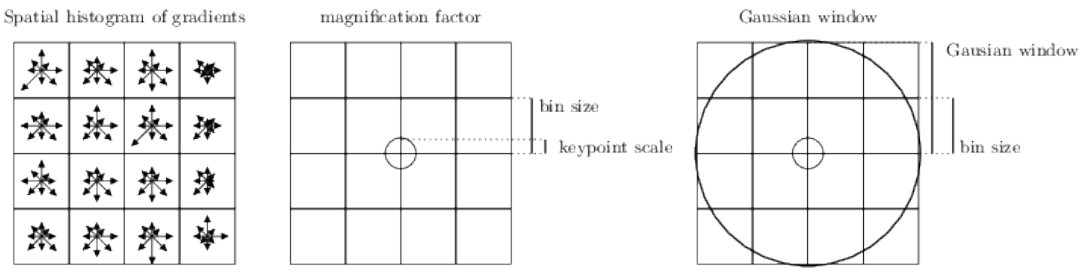


FIGURE 5.8: An overview of the SIFT Descriptor [19].

This histogram is, finally, the SIFT descriptor.

5.2.2 Feature Matching with FLANN

In order to match features between two image frames, the FLANN algorithm is defined below. This can be used in order to capture the motion of a camera during a video, just like in the following example and demonstration of visual odometry.

Classical feature descriptors like SURF or ORB are usually distinguished and matched using the Euclidean distance or L2-norm. Others use the Hamming distance. To filter the matches, David G. Lowe proposed to use a distance ratio test to try to eliminate false matches. **The distance ratio between a candidate keypoint's two nearest matches keypoint is computed** and it is considered an acceptable match when this metric is below a characteristic threshold. Indeed, this ratio allows for helping discriminate between ambiguous matches and well-discriminated matches. **FLANN stands for Fast Library for Approximate Nearest Neighbors.** It consists of a big compilation of optimized and ready algorithms for **fast nearest neighbor search**, as mentioned above, in large datasets and for high dimensional features, which work better in most cases than a simple Brute Force Matcher.

It is developed by the UBC Department of Computer Science in Vancouver, Canada. The interested reader could consult the [14] for an analytical review.

5.2.3 Experiment #3: Visual Odometry

Visual Odometry (VO) implies the means of incrementally estimating the pose of the vehicle by examining the changes that motion induces on the images of its onboard cameras. Some of VO's advantages are that it is not affected by adverse conditions like wheel slip in uneven terrains or rainy weathers. This produces more accurate trajectory estimates compared to wheel odometry. Its disadvantages include that in real life, we need an external sensor to estimate the absolute scale. Also, cameras may be cheap, but they are passive, which means that they might not be very robust when considering illumination changes and not only.

The formulation of the methodology, inspired by [24] is as below:

Algorithm 2: The Camera Motion Estimation Algorithm

Result: R, t for each frame

for *each subsequent frame* I_{k-1} *and* I_k **do**
    Apply SIFT feature descriptor for key points
    Use FLANN Feature Matcher to match features f_{k-1} and f_k
    Filter by distance if too many
    Find and decompose Essential Matrix (cv.findEssentialMat)
    Find rotation and translation matrices (cv.RecoverPose)
    Stack results to the inverse transformation matrix
end
Visualize the camera trajectory

Starting, we apply the SIFT descriptor in every image frame that we exported from CARLA, as mentioned in Chapter 2.

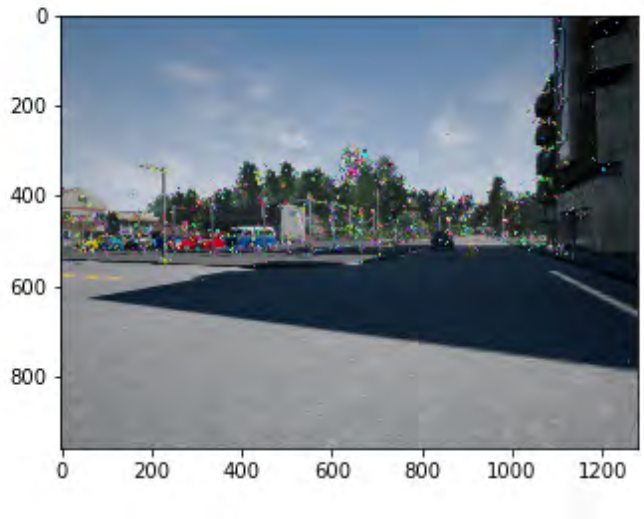


FIGURE 5.9: CARLA Image frame 1

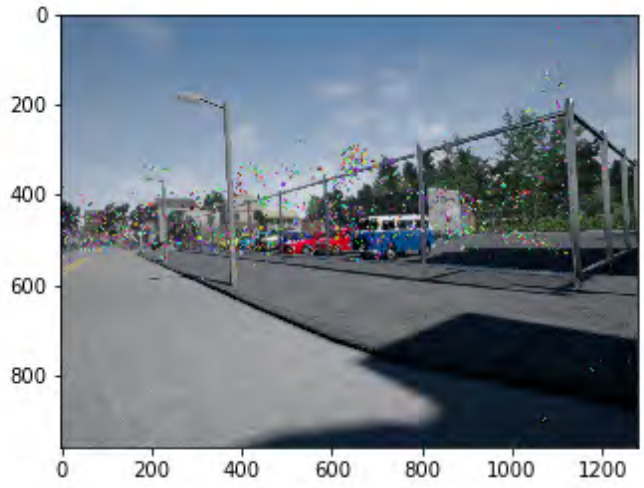


FIGURE 5.10: CARLA Image frame 2

Then, for each subsequent frame, we apply the FLANN Feature matcher.

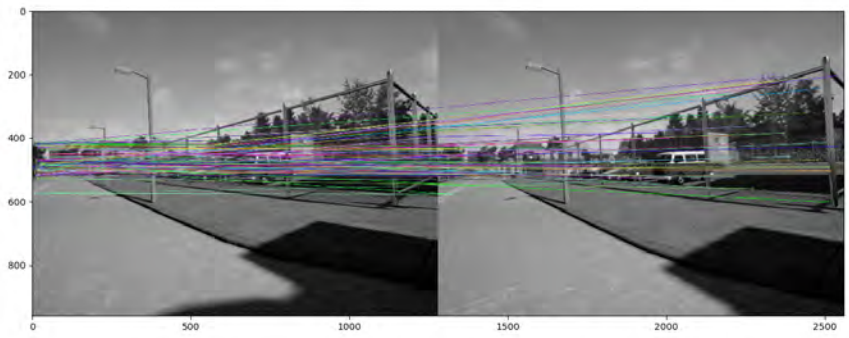


FIGURE 5.11: FLANN-based Matching Frame 1 (Unfiltered)

The matches are too many and confusing, so filtering is applied with distance as a metric between the best matches. In more detail, we apply a 0.6 distance threshold and choose to visualize only 25 matches, thus producing the following:

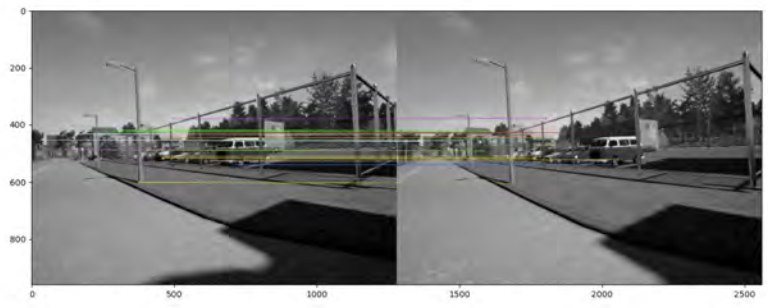


FIGURE 5.12: FLANN-based Matching Frame 1(Filtered)

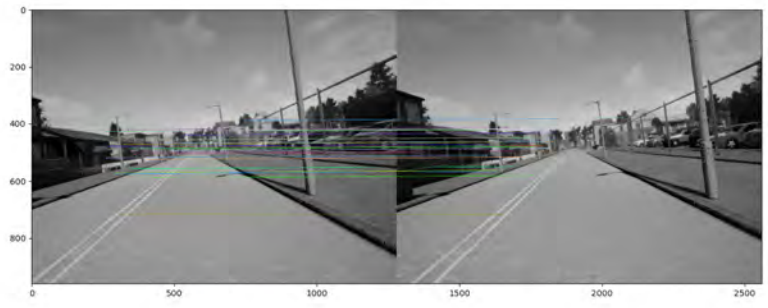


FIGURE 5.13: FLANN-based Matching Frame 2 (Filtered)

Then, for each pair of subsequent image frames, we try to estimate the camera motion. This is done by providing the first image points, the second image points and the camera calibration matrix *k* to OpenCV's *findEssentialMat* function [1], which returns the Essential Matrix E that was used between these two frames. Under the hood, this function provides an estimation of the essential matrix using a five-point algorithm. Followingly, we can provide the Essential Matrix E to OpenCV's *recoverPose* function [1], which determines the rotation and translation matrix for these two frames. In short, this is done by performing an SVD of the Essential matrix and verifying the solutions.

The main algorithm is shown in Python code below.

```
# Visual Odometry

for image in images:
    sift = cv.xfeatures2d.SIFT_create() # Apply SIFT
    kp, des = sift.detectAndCompute(image, None)

    kp_list.append(kp)
    des_list.append(des)

    # Visualize images with OpenCV
    display = cv.drawKeypoints(image, kp, None)
    plt.imshow(display)
```

```
for each subsequent image: # Find matches in image1 and image2

# For the sake of clarity,
# there is no filtering applied while the matching technique are
    simplified.

    # Config as stated in OpenCV
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 15)
    search_params = dict(checks=50)   # or pass empty dictionary

    flann = cv.FlannBasedMatcher(index_params,search_params)
    image1_points, image2_points = flann.knnMatch(des1,des2,k=2)

    E, _ = cv.findEssentialMat(np.array(image1_points), np.array(
    image2_points), dataset_handler.k, method = cv2.RANSAC)

    _, R, t, _ = cv.recoverPose(E, np.array(image1_points), np.array(
    image2_points), dataset_handler.k)
```

A demonstration, using CARLA's data inside a Jupyter Notebook environment, is shown below so we can validate if the results are correct by visualizing the motion between the two frames.

FIGURE 5.14: Keypoints motion visualization in a frame.

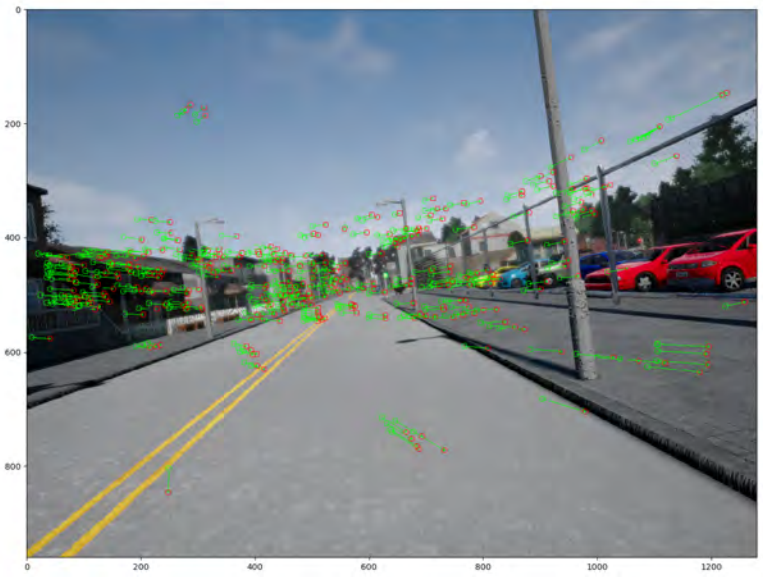


FIGURE 5.15: Keypoints motion visualization in a second frame.

The red circles define the key points location in the first frames and the green circles their location in the second frame.

Having the rotation and translation matrices for each consequent frame, we can then compute the camera transformation matrix which will provide us the camera motion in XYZ positions:

$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R^{-1} & -R^{-1}t \\ 0 & 1 \end{bmatrix} \tag{5.10}$$

Visualizing in a 3D figure, the camera motion estimated will be:

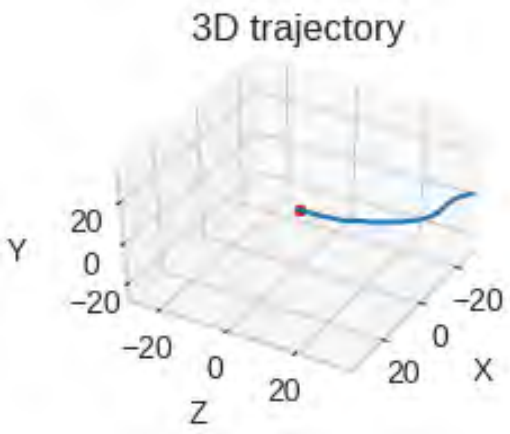


FIGURE 5.16: Camera motion estimation

The above figure seems to represent the motion of the original frame in an accurate and smooth figure, validating the original CARLA camera frames successfully. Inevitably, the X, Y, Z axes refer to the local space and not the global one, so we can only have an empirical sense of the result instead of comparing it to the original positions with metrics just like in the previous chapters.

5.2.4 Canny Edge Detector

The Canny edge detector is an operator that uses an algorithm of many stages to detect edges in images. It was developed in 1986 by John Canny.

The Canny edge detection algorithm consists of the following stages:

1. Noise reduction;
2. Gradient calculation;
3. Non-maximum suppression;
4. Edge Tracking by Hysteresis Thresholding

FIGURE 5.17: Canny Edge Detection Example [17]

Noise Reduction

The noise in an image affects the edges of it. Due to that, the first step is to remove the noise in the image with a Gaussian kernel, having a height and a depth of five. The filter size depends on the expected effect. The smallest the kernel, the less visible is the blur.

Gradient calculation

The processed image, which looks smoother, is then filtered with a Sobel filter in both horizontal and vertical direction to get accordingly the derivatives (Gx) and (Gy).

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2} Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{5.11}$$

Non-maximum suppression

Then, a full scan of the image pixels is done to remove any pixels that do not contribute to the form of an edge. This happens with the process of non-maximum suppression, where, every pixel needs to be local maximum in its defined neighborhood. The result is the same image with better and thinner edges. [17]

Edge Tracking by Hysteresis Thresholding

This stage is the one that decides which of the edges are edges and which are not. Two threshold values are needed, a minimum and a maximum one. The edges with a gradient higher than the *max* one are considered edges. Similarly, edges with a lower gradient than the *min* one are considered no edges. The edge with a *min* < gradient < *max* value is considered one of the two classes according to their neighborhood pixels.

Finally, the returned result is the sharp edges in the image. An open-source implementation of the Canny Edge Detector is provided through OpenCV [1], which we will use later in Experiment #4.2.

5.2.5 Hough Line Transform

The Hough transform is a feature extractor used in computer vision and digital image processing. The purpose of the Hough transform is to find imperfect instances of objects within a particular class of shapes. This is done with the help of voting. An edge detection technique, like the Canny Edge Detector, should be applied first as a pre-processing step [18].

A line can be expressed with two variables. For example, in the famous Cartesian coordinate system that would be (m, b). For Hough Transform, we will express lines in the Polar system. Henceforth, an equation for a line can be composed as:

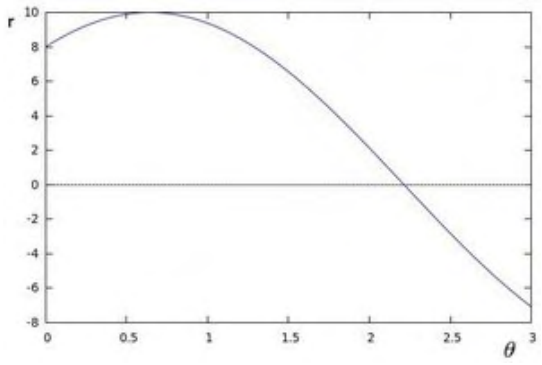
$$y = (-\frac{cos\theta}{sin\theta}x + \frac{r}{sin\theta}) \tag{5.12}$$

Arranging the terms: $r = xcos\theta + ysin\theta$ general for each point (x_0, y_0), we can characterize the group of lines that experiences that point as:

$$r_\theta = x_0 cos\theta + y_0 sin\theta \tag{5.13}$$

Meaning that each pair $r_\theta, \theta)$ represents each line that passes by (x_0, y_0)

If for a given (x_0, y_0) we plot the family of lines that goes through it, we get a sine wave.



We can do a similar activity as above for every one of the point in a picture. If the curves of two distinct points oscillate in the plane $\theta - r$, that means that both points belong to the same line [18] .

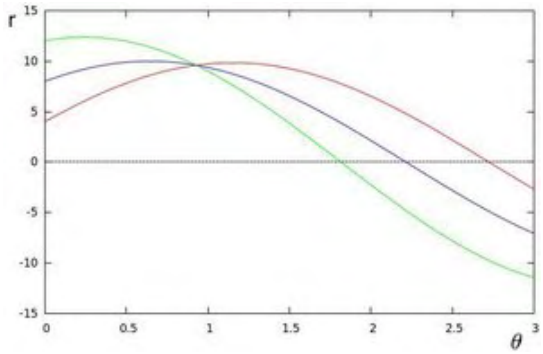


FIGURE 5.18: The lines intersection that is being exploited by the Hough transformation [18]

This means that in general, a line can be detected by finding the number of intersections between curves. Having more curves intersecting means that the line represented by that intersection have more points. In general, we can define a threshold as the base number of convergences/intersections expected to distinguish a line. [18]

So, the Hough Lines Estimator monitors the convergence between bends of each point in the picture. If the quantity of crossing points is above some defined threshold, then it's states as a line with the parameters (θ, r_θ) of the intersection point. [18]

The OpenCV package [18] provides two implementations of the Hough Line Transforms.

1. The Standard Hough Transform, returning (θ, r_θ)
2. The Probabilistic Hough Line Transform, returning the indices of the detected lines (x0,y0,x1,y1)

5.3 Artificial Neural Networks

A Feedforward Neural Network defines a mapping from input x to output y as:

$$y = f(x;\theta)$$

An N layer FNN is represented as the function composition:

$$f(x;\theta) = f^N(f^{N-1}(...f^2(f^1x))) \tag{5.14}$$

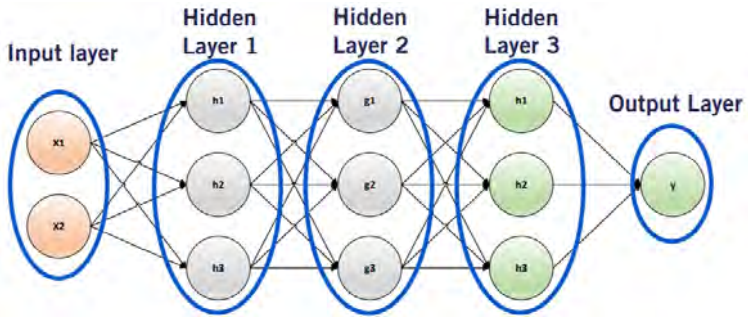


FIGURE 5.19: A neural network example [20].

An example of a deep neural network is designated above, where x is called the input layer, the functions f^1 to f^{N-1} are the hidden layers, and f^N is the output layer.

As a neural network is a machine learning methodology, its outputs are of two kinds: regression or classification. In a higher level, deep neural networks can be used for many tasks in the self-driving cars industry like object classification (image to a label), object detection (image to label and location), depth estimation (image to depth for every pixel) and semantic segmentation (image to label for every pixel).

Training

In order to train a neural network, examples of $f(x)$ are needed to be provided for a wide variation of the input x. Training can be done via presenting data in once (batch), in pieces (mini-batches) or even one by one.

Activation Function

A function f that defines the output of a single processing unit or neuron is called the activation function. Many different functions have been suggested over all these years (e.g., sigmoid, tanh), yet the most popular of all is the Rectified Linear Unit (ReLU), which expertly tackles the vanishing gradients problems.

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{5.15}$$

Loss Function

In optimization analysis, the function that is used to evaluate a possible solution is referred to as the objective or loss function. We may seek to maximize or minimize that loss function, implying that we are searching for a candidate solution that has the lowest or the biggest score among others. There are many types of loss functions which depend on the type of problem (regression, classification). Some of them include the Mean Squared Error, Mean Absolute Error, Categorical Cross-Entropy [20].

Optimization method

In order to find the minimum loss, different numerical analysis optimizers are being used. As stated above, their goal is to reduce the difference between the predicted output and the actual output. Such optimizers include Gradient Descent, Adagrad, and more. Also, the Adam optimization algorithm is a variant of the stochastic gradient descent method, famous and widely used in modern deep neural networks [20].

5.3.1 Convolutional Neural Networks

In a typical ANN, the final output would be

$$h_n = g(W^T h_{n-1} + b) \tag{5.16}$$

where b is the bias, h_{n-1} the outputs of the last layer, which are multiplied by the last weight vector W and then the activation function g is applied to it.

Convolutional Neural Networks have a different approach and exploit the use of convolutions between weights and hidden layers. CNNs contain some typical, fully connected layers at the end of each neural network model. They also still have a loss function on the last Fully-Connected layer. The main difference is that rather than learning unstructured weights, CNN architectures apply filters using convolution where the intrinsic values of the filters are the weights to be learned.

$$h_n = g(W^T * h_{n-1} + b) \tag{5.17}$$

where b is the bias, h_{n-1} the outputs of the last layer, which are convoluted with the last weight vector W and then the activation function g is applied to it.

Convolutions are computed per block of pixels. Their purpose is to show the cross-correlation of the original image with the filter that it is being convoluted with. Each filter is convolved over the height and the width of the image input, computing the inner product between the filter and the input and producing a 2D activation map. The neural network, as a result, learns filters that activate when it detects a specific feature type at some dimensional position of the image input. These operations are being calculated for each channel or depth of the original image. For example, RGB images contain three channels when greyscale images may contain only one channel.

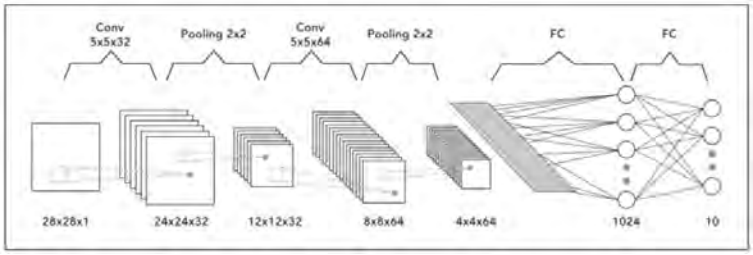


FIGURE 5.20: A convolutional neural network [20].

Pooling

Pooling is a form of a non-linear downsampler. Max-pooling is one widely used downsampling technique. It divides an image into a set of rectangles and, for each such region, it keeps the maximum and suppresses all the others.

A feature's location might be considered valuable when it is relative to other such features or keypoints, in contrast to its precise location. This is also the idea behind the vast use of pooling in CNNs. The pooling layer serves to diminish the size of the image representation progressively, to reduce the number of parameters that are being used in a vast neural network, like a convolutional one.

All the remaining properties of a typical artificial neural network remain the same, like optimization, loss function, regularization, and more. CNNs are used for object detection, semantic segmentation, and generally any machine learning task (regression or classification). They show their most significant potential when working with image datasets.

5.4 Object Detection

There have been many algorithms used throughout these years that could detect objects in a 2D image with a large success rate. Such efforts include the Viola Jones Object Detection Framework, the Histogram of Oriented Gradients but the big change in the scene came with a convolutional neural network, called AlexNet, in 2012.

CNNs and the era of deep learning changed this sector of applications in a wide manner. Object Detection in the area of driverless vehicles could look like this:

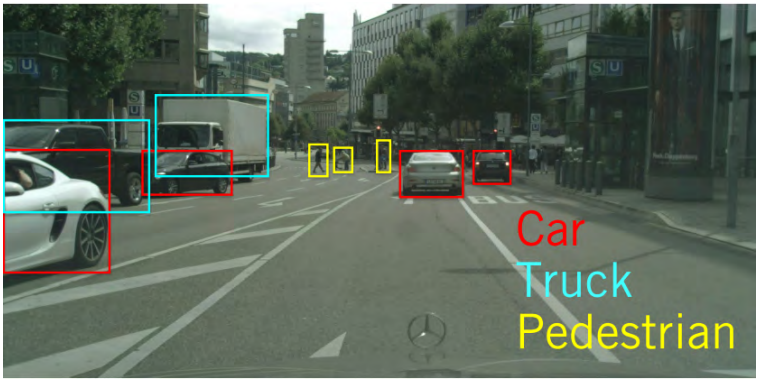


FIGURE 5.21: An object detection example for driverless vehicles [24].

We want the neural network's output to draw a bounding box in the screen, defining the x, y positions of the possible object. The category of objects is determined by the training data that is provided. If images with labeled data of K classes are provided, then, the desired output would look like:

$$f(x;\theta) = [x_{min}, y_{min}, x_{max}, y_{max}, S_{class1}, ..., S_{classK}] \tag{5.18}$$

ConvNets can be used in order to map a 2D image into the desired output:

FIGURE 5.22: A ConvNet for object detection [24].

Many problems occur in the task of object detection. For example, the extent of objects is not entirely observed many times due to occlusion or truncation. Besides, illumination changes can also happen, making the image too bright or too dark. Lastly, a challenging task is object detection in large and high-resolution data. Convolutions are a quite expensive operation, thus, making this research area a lot challenging in order to achieve low-cost efficiency.

Famous modern-day algorithms that perform the task of 2D Object Detection successfully include the "You Only Look Once" (YOLO) Algorithm and the R-CNN, which may be a bit expensive computationally.

5.5 Semantic Segmentation

Semantic segmentation is the task of recognizing and understanding what in an image in pixel level. It is a very challenging area of machine learning, and many methodologies are introduced every day [24].

It is used for robot vision and understanding, and it is especially useful in autonomous driving. Mathematically, it looks similar to object detection

$$f(x;\theta) = [S_{class1}, ..., S_{classK}] \tag{5.19}$$

where S denotes the probability for each class. Such classes may contain cars, traffic light, and signs, sidewalk.

FIGURE 5.23: A Semantic Segmentation example [24].

Just like object detection, semantic segmentation is not trivial. Occlusion, truncation, scale, and illumination changes can happen. Another level of difficulty, if we think at a low level, is that smooth boundaries are needed. This means that semantic segmentation needs highly accurate results. Convolutional Neural Networks, once again, show an outstanding benchmark performance on this kind of tasks.

Mathematically, the task of semantic segmentation is equal to a multiclass classification for every pixel. It consists of providing a class label for every pixel in a 2D image. Many methods exist in order to achieve better performance, such as evaluation using class Intersection-over-Union. Popular neural networks for this task is the DeepLab Net, the SegNet.

5.6 Experiment #4: Road Scene Understanding with the use of a Neural Network

The CARLA Simulation environment has a built-in segmentation neural network. As explained in Chapter 3, it can do multiclass classification in pixel level and provide its outputs in an API. This information will be used in order to be fused with some computer vision techniques to showcase how road scene understanding can be done for an autonomous vehicle.

5.6.1 Experiment #4.1: 3D Drivable Surface Estimation with RANSAC

In this experiment, we are going to estimate the drivable surface for a smart car using the output from CARLA's semantic segmentation neural network. As our dataset, we have 3 Image objects from CARLA's simulation environment, as explained in Chapter 2, which are depicted below. All of them will be used in order to fine tune the arbitrary thresholds.

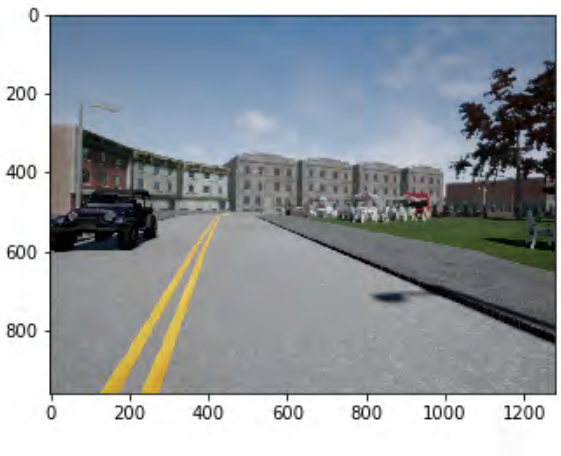


FIGURE 5.24: CARLA's Exported Image 1

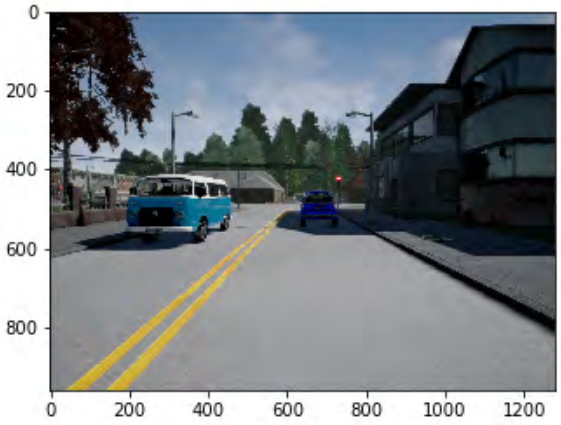


FIGURE 5.25: CARLA's Exported Image 2

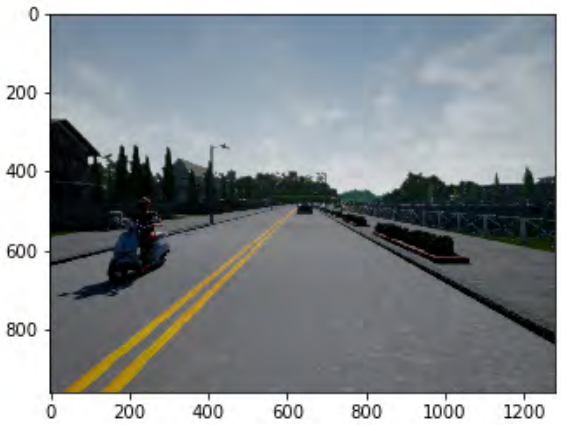


FIGURE 5.26: CARLA's Exported Image 2

An example of how CARLA's segmentation looks like is illustrated below.

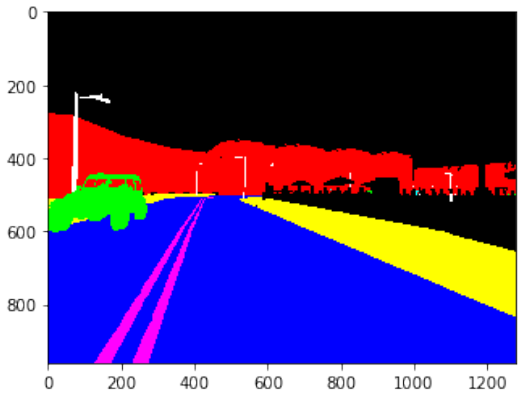


FIGURE 5.27: A Segmentated Image by CARLA's Neural Network [3]

In order to express the picture in a 3D frame (the camera's reference frame), we will exploit the transformation used in previous subsections as long as the CARLA's depth sensor.

$$z = depth \tag{5.20}$$

$$x = \frac{(u - c_u) * z}{f} \tag{5.21}$$

$$y = \frac{(v - c_v) * z}{f} \tag{5.22}$$

where c_u, c_v are the camera's centers, and u, v the pixel positions. The camera's focal length is denoted by f. All of them are called the camera's intrinsic parameters.

Having the frame expressed in 3D XYZ format, we can implement the RANSAC algorithm for plane estimation.

RANSAC - Random Sample Consensus

Random sample consensus (RANSAC) is an iterative approach for estimating a mathematical model and its goal is to reject outliers from a dataset. The RANSAC approach works by determining the dataset's outliers set and estimating the desired model using inlier data [12].

RANSAC can be implemented with the following steps:

1. Select a random subgroup from the dataset
2. Fit a model to the selected subgroup
3. Determine the number of outliers
4. Repeat previous steps for a determined number of iterations

The proposed methodology for the drivable surface estimation, which makes use of the RANSAC algorithm, is defined as:

Algorithm 3: RANSAC for Ground Plane Fitting

Result: Returns a plane model

for *iterations (N=100) or minimum inliers (M=10000)* **do**
- Mask image with CARLA's ground class
- Choose a minimum N (N=15) of 3 points from xyz_{ground} at random.
- Compute the ground plane model $ax + by + cz + d = 0$ using SVD with the chosen random points
- Calculate the distance from the ground plane model to every point in xyz_{ground} $(ax + by + cz + d)/\sqrt{a^2 + b^2 + c^2}$
- Compute the number of inliers based on a distance threshold (0.3m)
- Keep the inlier set with the most significant number of points.

end

Recompute and return a plane model using all inliers in the final inlier set

For the solution of the linear system for the plane, we used the NumPy [16] package and specifically the Singular-Value Decomposition (SVD) [5] function. Other numerical analysis methods could be exploited.

```
# RANSAC For Plane Fitting
max_inliers_cnt = 0
max_inliers_set_idx = None

# Set thresholds:
num_itr = 1000  # RANSAC maximum number of iterations
min_num_inliers = 10000  # RANSAC minimum number of inliers
distance_threshold = 0.3  # Maximum distance from point to plane for point
    to be considered inlier

for i in range(num_itr):
# Step 1: Choose a minimum of 3 points from xyz_data at random.
    idx = np.random.choice(range(xyz_data.shape[1]), 15, replace=False) #
    Choose 15 random points

    # Step 2: Compute plane model
    plane_param = compute_SVD_for_plane(xyz_data[:, idx])

    # Step 3: Find number of inliers
    distances = dist_to_plane(plane_param.T, xyz_data[0, :].T, xyz_data[1,
     :].T, xyz_data[2, :].T)

    # Step 4: Check if the current number of inliers is greater than all
    previous iterations and keep the inlier set with the largest number of
    points.
    inliers_cnt = np.sum(distances < distance_threshold)

    if inliers_cnt > max_inliers_cnt:
```

```
        max_inliers_cnt = inliers_cnt
        max_inliers_set_idx = np.where(distances < distance_threshold)[0]

    # Step 5: Check if stopping criterion is satisfied and break.
    if inliers_cnt > 10000 or i > num_itr:
        break

    # Step 6: Recompute the model parameters using largest inlier set.
    final_data = xyz_data[:, max_inliers_set_idx]
    output_plane = compute_plane(final_data)
```

The final estimated plane model is visualized below:

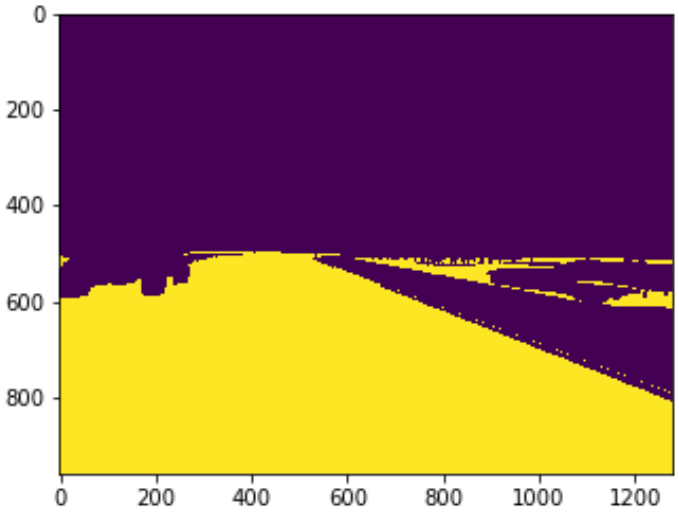


FIGURE 5.28: Visualized inliers after the proposed RANSAC + SVD methodology.

A 3D representation of the above model is also provided:

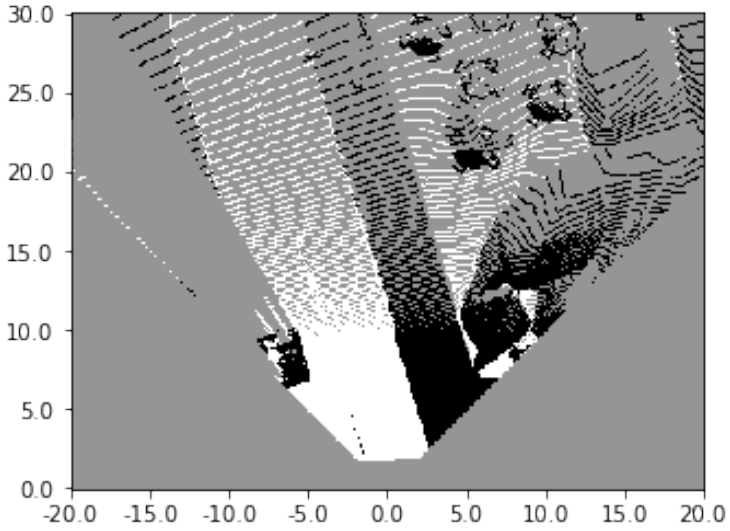


FIGURE 5.29: A 3D space representation of visualized inliers.

Notice that the black pixels here belong to areas not belonging to the estimated ground class. Also, The black pixels on the left represent the incoming vehicle.

The estimated drivable surface looks good for most of the image. Notice that there are some outliers on the right instead of the sidewalk. Other than that, the RANSAC algorithm is pretty satisfying and can show to a self-driving car the ground that it can drive on. However, a car needs to follow its lane and not invade other lanes. That is why the location of its lanes is needed to be known.

5.6.2 Experiment #4.2: Semantic Lane Estimation

In order for a driverless vehicle to track its lane, we are going to use once again the outputs from CARLA's Semantic Segmentation neural network. Specifically, the methodology is explained below:

Algorithm 4: Semantic Lane Estimation

Result: Return road lanes of a car

1 Mask image with CARLA's road lanes and sidewalks class
2 Extract edges using the Canny Edge Detector
3 Perform the Hough Lines Estimator
4 Filter unnecessary lanes (e.g. outliers or horizontal lanes)

The main Python code is shown below in order to be reproduced.

```
# Lane Estimation
 # Step 1: Mask the pixels of the image that belong to lane boundary
    categories from the output of semantic segmentation

lane_mask = (segmentation==6) | (segmentation==8)
```

```
# Step 2: Perform Edge Detection using cv.Canny()
mask_canny = cv.Canny(lane_mask * 255, 50, 100)

# Step 3: Perform Line estimation using cv.HoughLinesP()
lines = cv.HoughLinesP(mask_canny, rho=10, theta=np.pi/180*1, threshold
    =100, minLineLength=200, maxLineGap=100)
```

After applying masking, edge detection, and lines estimation, we get the following visual results:

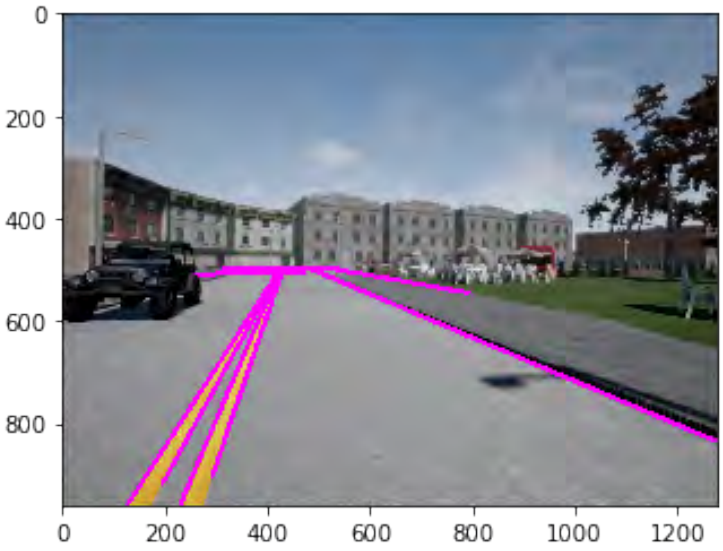


FIGURE 5.30: The Semantic Lane Estimation (Unfiltered)

Few outliers can be seen, such as the far top left line and the park side lane. A slope threshold can filter these. The slope for each line can be estimated geometrically. In mathematical language the slope m of the line is

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{5.23}$$

We apply an arbitrary minimum slope threshold of 0.3, thus resulting in the following:

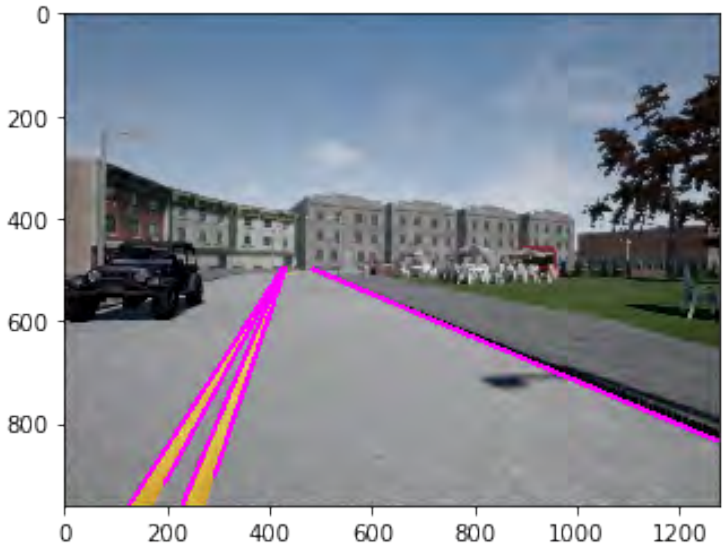


FIGURE 5.31: The Semantic Lane Estimation (Filtered)

The estimation now looks correct. This can be provided to a Robotics Controller or a motion planning algorithm and inform the car on how to follow the lane (e.g., how much speed, drift).

5.6.3 Experiment #4.3: Computing Minimum Distance to Impact: A Collision System

A driverless vehicle has safety as the number one priority. It cannot trust a single semantic segmentation network for its classification, neither an object detection algorithm, when concerning human lives. Sometimes, even unique algorithms like YOLOv3 or R-CNN may show high recall and low precision on their classification. When it comes to detecting danger (e.g., other cars, motorcycles or pedestrians incoming to impact), different systems' results are used and fused to final output. Just like in the case of localization where we could not trust only one sensor, here, we will exploit the fusion of two outputs, one output that comes from semantic segmentation, and another one that may come from a faulty object detection algorithm.

In the next frame, one car is shown in the scene. No pedestrians, bikes, or motorcycles are in the image. However, an object detection algorithm could showcase three possible anchor boxes for vehicles. Visually, we can understand that this is faulty and that the two examples are False Positives from our algorithm.

FIGURE 5.32: Unreliable results in object detection

First, there is the task of eliminating the False Positives and then computing the Euclidean of the foreign object to CARLA's car.

Our proposed methodology for the False Positives elimination includes the fusion from the segmentation neural network and the object detection algorithm outputs. Of course, a square box or anchor box is defined by its four edges, $x_{min}, y_{min}, x_{max}, y_{max}$

Specifically:

Algorithm 5: Filter out Unreliable Detections

Result: True object detection

```
for each possible detection do
    Compute how many pixels in the bounding box belong to the category
     predicted by CARLA's neural network
    Divide the computed number of pixels by the area of the bounding box (total
     number of pixels)
    if ratio > threshold (0.3) then
        keep the detection;
    else
        remove the detection;
    end
end
```

After implementing the algorithm in Python, we have the following output:

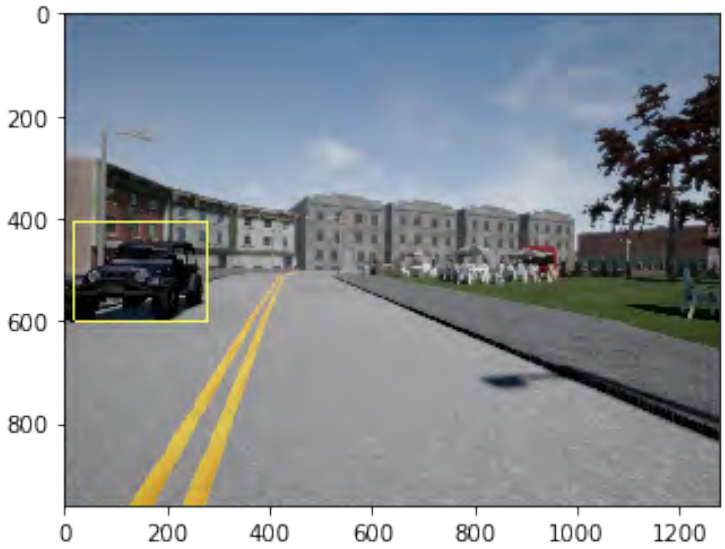


FIGURE 5.33: The result of filtering out object detection's unreliable results with the use of semantic segmentation

After keeping only the true positives of the fusion from the two systems, we are interested in calculating the minimum distance to it.

This can be easily calculated by calculating the Euclidean distance ($\sqrt{x^2 + y^2 + z^2}$) for each pixel of the true bounding box since the image is already expressed in the 3D Camera reference frame. Then, we can scan and keep only the minimum distance (since different parts of the object have a different depth, hence, different distance from the camera).

The resulting image is below:

FIGURE 5.34: A showcase of calculating the distance until impact with the use of 3D image representation

This could help, e.g., as a danger notification to the motion planning algorithm that the vehicle uses. For example, if the distance to collision is lower than 1m, steer to the other side. These types of input signals are provided to motion planning algorithms and robotic operating systems. A great operating system is *ROS*, available at ros.org. CARLA provides a bridge library for connecting CARLA with ROS, but it is still in early development. As it is outside of the scope of this thesis, it would be considered attractive as future work.

Chapter 6

Conclusion

6.1 Summary

We have presented many ways to approach the topics of vehicle automation, state estimation, and various tasks of visual perception under the context of self-driving cars. In more detail, in Chapter 3, we proposed the PID and the Stanley controller for longitudinal and lateral control with successful results. Next, in Chapter 4, we investigated the use of the Extended Kalman Filter for sensor fusion with satisfying results in the task of a vehicle's localization. Chapter 5 shows how visual odometry can be achieved in order to extend localization while it also presents how segmentation neural networks can be combined with different computer vision algorithms to accomplish accurate road scene understanding, achieving acceptable results.

6.2 Future work

First of all, the study on CARLA's datasets could be completed by experimenting with motion planning algorithms. There are also many individuals that try to combine all of the above with Deep Reinforcement Learning techniques.
I would also suggest to any interested persons that would like to dive in the Self-Driving Cars business to explore Udacity's related Nanodegree [23] or the courses from University of Toronto [25], which inspired me in many ways to modify and create the path for this thesis.

More specifically, different controllers could be implemented and compared in Chapter 3 such as the Pure Pursuit Controller or the Model-Predictive Controller (MPC). Furthermore, more Bayesian approaches could be exploited in Chapter 4 such as the Unscented Kalman Filter, the Indirect Kalman Filter or Particle Filtering. Besides, it would be considered useful to perform the proposed approaches of Chapter 5 to real datasets, like KITTI [4] or commaai's *(commaai.com)* where there are worn out lanes and sidewalks, something that is not yet implemented in the CARLA Simulation environment The interested reader or researcher could also integrate the above

into a Robot Operating Systems (ROS) and evaluate these practices in a real robot. The writer welcomes any attempts for contribution and is available for any assistance through their GitHub account, where the code is uploaded.

Bibliography

[1] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[2] Computer Science & Engineering, Washington. *EE/CSE 575: Computer Vision*. 2018. URL: https://courses.cs.washington.edu/courses/cse576/18sp/notes/index.html.

[3] Alexey Dosovitskiy et al. "CARLA: An open urban driving simulator". In: *arXiv preprint arXiv:1711.03938* (2017).

[4] Andreas Geiger et al. "Vision meets robotics: The KITTI dataset". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1231–1237.

[5] Gene H Golub and Christian Reinsch. "Singular value decomposition and least squares solutions". In: *Linear Algebra*. Springer, 1971, pp. 134–151.

[6] Instrumentationforum. *Instrumentationforum website*. [Online; accessed 20-May-2019]. 2018. URL: https://instrumentationforum.com/t/pid-controller-manual-tuning/4043.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[8] Roger Labbe. "Kalman and bayesian filters in python". In: (2015).

[9] Levelfivesupplies. *Levelfivesupplies website*. 2019. URL: https://levelfivesupplies.com/sensors-used-in-autonomous-vehicles/.

[10] Guosheng Lin et al. "Refinenet: Multi-path refinement networks for high-resolution semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1925–1934.

[11] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[12] MathWorks. *Using RANSAC for estimating geometric transforms in computer vision*. URL: https://www.mathworks.com/discovery/ransac.html.

[13] MIT. *MIT 6.S094: Deep Learning for Self-Driving Cars*. 2019. URL: https://selfdrivingcars.mit.edu.

[14] Marius Muja and David Lowe. "Flann-fast library for approximate nearest neighbors user manual". In: *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).

[15] National Instruments (ni.com). *PID Theory Explained*. 2019. URL: http://www.ni.com/en-my/innovations/white-papers/06/pid-theory-explained.html.

[16] Travis Oliphant. *NumPy: A guide to NumPy*. USA: Trelgol Publishing. [Online; accessed <today>]. 2006–. URL: http://www.numpy.org/.

[17] OpenCV. *Canny Edge Detection*. URL: https://docs.opencv.org/master/da/d22/tutorial_py_canny.html.

[18] OpenCV. *Hough Line Transform*. URL: https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html.

[19] OpenCV. *Introduction to SIFT (Scale-Invariant Feature Transform)*. URL: https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html.

[20] Stanford University. *CS230 Deep Learning*. 2019. URL: https://cs230.stanford.edu/.

[21] Sebastian Thrun et al. "Stanley: The robot that won the DARPA Grand Challenge". In: *Journal of field Robotics* 23.9 (2006), pp. 661–692.

[22] Towards Data Science. *An intro to Kalman Filters for Autonomous Vehicles*. 2019. URL: https://towardsdatascience.com/an-intro-to-kalman-filters-for-autonomous-vehicles-f43dd2e2004b.

[23] Udacity. *Self-Driving Car Nanodegree*. 2019. URL: https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013.

[24] University Of Toronto. *CSC2541: Visual Perception for Autonomous Driving*. 2016. URL: http://www.cs.toronto.edu/~urtasun/courses/CSC2541/CSC2541_Winter16.html.

[25] University Of Toronto. *Self-Driving Car Specialization*. 2019. URL: https://www.coursera.org/specializations/self-driving-cars.

[26] University Of Toronto. *Self-Driving Cars*. 2019. URL: https://www.coursera.org/lecture/state-estimation-localization-self-driving-cars/why-sensor-fusion-HCP35.

[27] WaveLab. *Autonomous Vehicles Laboratory, University of Waterloo*. URL: http://wavelab.uwaterloo.ca/.

[28] Wikipedia contributors. *Kalman filter — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-May-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Kalman_filter&oldid=897230289.

[29] Wikipedia contributors. *PID controller — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-May-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=PID_controller&oldid=897360228.